rdflib Documentation

Release 4.2.1

RDFLib Team

March 02, 2017

Contents

RDFLib is a pure Python package work working with RDF. RDFLib contains most things you need to work with RDF, including:

- parsers and serializers for RDF/XML, N3, NTriples, N-Quads, Turtle, TriX, RDFa and Microdata.
- a Graph interface which can be backed by any one of a number of Store implementations.
- store implementations for in memory storage and persistent storage on top of the Berkeley DB.
- a SPARQL 1.1 implementation - supporting SPARQL 1.1 Queries and Update statements.

CHAPTER 1

Getting started

If you never used RDFLib, click through these

Getting started with RDFLib

Installation

RDFLib is open source and is maintained in a GitHub repository. RDFLib releases, current and previous are listed on PyPi

The best way to install RDFLib is to use `easy_install` or `pip`:

```
$ easy_install rdflib
```

Support is available through the rdflib-dev group:

> http://groups.google.com/group/rdflib-dev

and on the IRC channel #rdflib on the freenode.net server

The primary interface that RDFLib exposes for working with RDF is a *`Graph`*. The package uses various Python idioms that offer an appropriate way to introduce RDF to a Python programmer who hasn't worked with RDF before.

RDFLib graphs are not sorted containers; they have ordinary `set` operations (e.g. `add()` to add a triple) plus methods that search triples and return them in arbitrary order.

RDFLib graphs also redefine certain built-in Python methods in order to behave in a predictable way; they emulate container types and are best thought of as a set of 3-item triples:

```
[
    (subject,  predicate,  object),
    (subject1, predicate1, object1),
    ...
    (subjectN, predicateN, objectN)
 ]
```

A tiny usage example:

```
import rdflib

g = rdflib.Graph()
result = g.parse("http://www.w3.org/People/Berners-Lee/card")

print("graph has %s statements." % len(g))
```

```
# prints graph has 79 statements.

for subj, pred, obj in g:
   if (subj, pred, obj) not in g:
       raise Exception("It better be!")

s = g.serialize(format='n3')
```

A more extensive example:

```
from rdflib import Graph, Literal, BNode, Namespace, RDF, URIRef
from rdflib.namespace import DC, FOAF

g = Graph()

# Create an identifier to use as the subject for Donna.
donna = BNode()

# Add triples using store's add method.
g.add( (donna, RDF.type, FOAF.Person) )
g.add( (donna, FOAF.nick, Literal("donna", lang="foo")) )
g.add( (donna, FOAF.name, Literal("Donna Fales")) )
g.add( (donna, FOAF.mbox, URIRef("mailto:donna@example.org")) )

# Iterate over triples in store and print them out.
print("--- printing raw triples ---")
for s, p, o in g:
    print((s, p, o))

# For each foaf:Person in the store print out its mbox property.
print("--- printing mboxes ---")
for person in g.subjects(RDF.type, FOAF.Person):
    for mbox in g.objects(person, FOAF.mbox):
        print(mbox)

# Bind a few prefix, namespace pairs for more readable output
g.bind("dc", DC)
g.bind("foaf", FOAF)

print( g.serialize(format='n3') )
```

Many more examples can be found in the `examples` folder in the source distribution.

Loading and saving RDF

Reading an NT file

RDF data has various syntaxes (`xml`, `n3`, `ntriples`, `trix`, etc) that you might want to read. The simplest format is `ntriples`, a line-based format. Create the file `demo.nt` in the current directory with these two lines:

```
<http://bigasterisk.com/foaf.rdf#drewp> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://xml
<http://bigasterisk.com/foaf.rdf#drewp> <http://example.com/says> "Hello world" .
```

You need to tell RDFLib what format to parse, use the `format` keyword-parameter to *`parse()`*, you can pass either a mime-type or the name (a list of available parsers is available). If you are not sure what format your file will be, you can use *`rdflib.util.guess_format()`* which will guess based on the file extension.

In an interactive python interpreter, try this:

```
from rdflib import Graph

g = Graph()
g.parse("demo.nt", format="nt")

len(g) # prints 2

import pprint
for stmt in g:
    pprint.pprint(stmt)

# prints :
(rdflib.term.URIRef('http://bigasterisk.com/foaf.rdf#drewp'),
 rdflib.term.URIRef('http://example.com/says'),
 rdflib.term.Literal(u'Hello world'))
(rdflib.term.URIRef('http://bigasterisk.com/foaf.rdf#drewp'),
 rdflib.term.URIRef('http://www.w3.org/1999/02/22-rdf-syntax-ns#type'),
 rdflib.term.URIRef('http://xmlns.com/foaf/0.1/Person'))
```

The final lines show how RDFLib represents the two statements in the file. The statements themselves are just length-3 tuples; and the subjects, predicates, and objects are all rdflib types.

Reading remote graphs

Reading graphs from the net is just as easy:

```
g.parse("http://bigasterisk.com/foaf.rdf")
len(g)
# prints 42
```

The format defaults to `xml`, which is the common format for .rdf files you'll find on the net.

RDFLib will also happily read RDF from any file-like object, i.e. anything with a `.read` method.

Creating RDF triples

Creating Nodes

RDF is a graph where the nodes are URI references, Blank Nodes or Literals, in RDFLib represented by the classes *URIRef*, *BNode*, and *Literal*. `URIRefs` and `BNodes` can both be thought of as resources, such a person, a company, a web-site, etc. A `BNode` is a node where the exact URI is not known. `URIRefs` are also used to represent the properties/predicates in the RDF graph. `Literals` represent attribute values, such as a name, a date, a number, etc.

Nodes can be created by the constructors of the node classes:

```
from rdflib import URIRef, BNode, Literal

bob = URIRef("http://example.org/people/Bob")
linda = BNode() # a GUID is generated

name = Literal('Bob') # passing a string
age = Literal(24) # passing a python int
height = Literal(76.5) # passing a python float
```

Literals can be created from python objects, this creates `data-typed literals`, for the details on the mapping see *Literals*.

For creating many `URIRefs` in the same `namespace`, i.e. URIs with the same prefix, RDFLib has the *`rdflib.namespace.Namespace`* class:

```
from rdflib import Namespace

n = Namespace("http://example.org/people/")

n.bob # = rdflib.term.URIRef(u'http://example.org/people/bob')
n.eve # = rdflib.term.URIRef(u'http://example.org/people/eve')
```

This is very useful for schemas where all properties and classes have the same URI prefix, RDFLib pre-defines Namespaces for the most common RDF schemas:

```
from rdflib.namespace import RDF, FOAF

RDF.type
# = rdflib.term.URIRef(u'http://www.w3.org/1999/02/22-rdf-syntax-ns#type')

FOAF.knows
# = rdflib.term.URIRef(u'http://xmlns.com/foaf/0.1/knows')
```

Adding Triples

We already saw in Loading and saving RDF, how triples can be added with with the *`parse()`* function.

Triples can also be added with the *`add()`* function:

`Graph.`**`add`**`((s, p, o))`
Add a triple with self as context

`add()` takes a 3-tuple of RDFLib nodes. Try the following with the nodes and namespaces we defined previously:

```
from rdflib import Graph
g = Graph()

g.add( (bob, RDF.type, FOAF.Person) )
g.add( (bob, FOAF.name, name) )
g.add( (bob, FOAF.knows, linda) )
g.add( (linda, RDF.type, FOAF.Person) )
g.add( (linda, FOAF.name, Literal('Linda') ) )

print g.serialize(format='turtle')
```

outputs:

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xml: <http://www.w3.org/XML/1998/namespace> .

<http://example.org/people/Bob> a foaf:Person ;
        foaf:knows [ a foaf:Person ;
                     foaf:name "Linda" ] ;
        foaf:name "Bob" .
```

For some properties, only one value per resource makes sense (i.e they are *functional properties*, or have max-cardinality of 1). The *`set()`* method is useful for this:

```
g.add( ( bob, FOAF.age, Literal(42) ) )
print "Bob is ", g.value( bob, FOAF.age )
# prints: Bob is 42

g.set( ( bob, FOAF.age, Literal(43) ) ) # replaces 42 set above
print "Bob is now ", g.value( bob, FOAF.age )
# prints: Bob is now 43
```

rdflib.graph.Graph.value() is the matching query method, it will return a single value for a property, optionally raising an exception if there are more.

You can also add triples by combining entire graphs, see *Set Operations on RDFLib Graphs*.

Removing Triples

Similarly, triples can be removed by a call to *remove()*:

Graph.**remove**(*(s, p, o)*)
Remove a triple from the graph

If the triple does not provide a context attribute, removes the triple from all contexts.

When removing, it is possible to leave parts of the triple unspecified (i.e. passing None), this will remove all matching triples:

```
g.remove( (bob, None, None) ) # remove all triples about bob
```

An example

LiveJournal produces FOAF data for their users, but they seem to use foaf:member_name for a person's full name. To align with data from other sources, it would be nice to have foaf:name act as a synonym for foaf:member_name (a poor man's one-way owl:equivalentProperty):

```
from rdflib.namespace import FOAF
g.parse("http://danbri.livejournal.com/data/foaf")
for s,_,n in g.triples((None, FOAF['member_name'], None)):
    g.add((s, FOAF['name'], n))
```

Navigating Graphs

An RDF Graph is a set of RDF triples, and we try to mirror exactly this in RDFLib, and the graph tries to emulate a container type:

Graphs as Iterators

RDFLib graphs override *__iter__()* in order to support iteration over the contained triples:

```
for subject,predicate,obj in someGraph:
   if not (subject,predicate,obj) in someGraph:
      raise Exception("Iterator / Container Protocols are Broken!!")
```

Contains check

Graphs implement *__contains__()*, so you can check if a triple is in a graph with `triple in graph` syntax:

```
from rdflib import URIRef
from rdflib.namespace import RDF
bob = URIRef("http://example.org/people/bob")
if ( bob, RDF.type, FOAF.Person ) in graph:
   print "This graph knows that Bob is a person!"
```

Note that this triple does not have to be completely bound:

```
if (bob, None, None) in graph:
   print "This graph contains triples about Bob!"
```

Set Operations on RDFLib Graphs

Graphs override several pythons operators: *__iadd__()*, *__isub__()*, etc. This supports addition, subtraction and other set-operations on Graphs:

operation	effect
`G1 + G2`	return new graph with union
`G1 += G1`	in place union / addition
`G1 - G2`	return new graph with difference
`G1 -= G2`	in place difference / subtraction
`G1 & G2`	intersection (triples in both graphs)
`G1 ^ G2`	xor (triples in either G1 or G2, but not in both)

Warning: Set-operations on graphs assume bnodes are shared between graphs. This may or may not do what you want. See Merging graphs for details.

Basic Triple Matching

Instead of iterating through all triples, RDFLib graphs support basic triple pattern matching with a *triples()* function. This function is a generator of triples that match the pattern given by the arguments. The arguments of these are RDF terms that restrict the triples that are returned. Terms that are `None` are treated as a wildcard. For example:

```
g.load("some_foaf.rdf")
for s,p,o in g.triples( (None, RDF.type, FOAF.Person) ):
   print "%s is a person"%s

for s,p,o in g.triples( (None,  RDF.type, None) ):
   print "%s is a %s"%(s,o)

bobgraph = Graph()

bobgraph += g.triples( (bob, None, None) )
```

If you are not interested in whole triples, you can get only the bits you want with the methods *objects()*, *subjects()*, *predicates()*, `predicates_objects()`, etc. Each take parameters for the components of the triple to constraint:

```
for person in g.subjects(RDF.type, FOAF.Person):
   print "%s is a person"%person
```

Finally, for some properties, only one value per resource makes sense (i.e they are *functional properties*, or have max-cardinality of 1). The `value()` method is useful for this, as it returns just a single node, not a generator:

```
name = g.value(bob, FOAF.name)  # get any name of bob
# get the one person that knows bob and raise an exception if more are found
mbox = g.value(predicate = FOAF.name, object = bob, any = False)
```

`Graph` methods for accessing triples

Here is a list of all convenience methods for querying Graphs:

`Graph.`**`label`**(*subject*, *default=''*)

Query for the RDFS.label of the subject

Return default if no label exists or any label if multiple exist.

`Graph.`**`preferredLabel`**(*subject*, *lang=None*, *default=None*, *labelProperties=(rdflib.term.URIRef(u'http://www.w3.org/2004/02/skos/core#prefLabel'), rdflib.term.URIRef(u'http://www.w3.org/2000/01/rdf-schema#label'))*)

Find the preferred label for subject.

By default prefers skos:prefLabels over rdfs:labels. In case at least one prefLabel is found returns those, else returns labels. In case a language string (e.g., 'en', 'de' or even '' for no lang-tagged literals) is given, only such labels will be considered.

Return a list of (labelProp, label) pairs, where labelProp is either skos:prefLabel or rdfs:label.

```
    >>> from rdflib import ConjunctiveGraph, URIRef, RDFS, Literal
    >>> from rdflib.namespace import SKOS
    >>> from pprint import pprint
    >>> g = ConjunctiveGraph()
    >>> u = URIRef(u'http://example.com/foo')
    >>> g.add([u, RDFS.label, Literal('foo')])
    >>> g.add([u, RDFS.label, Literal('bar')])
    >>> pprint(sorted(g.preferredLabel(u)))
    [(rdflib.term.URIRef(u'http://www.w3.org/2000/01/rdf-schema#label'),
      rdflib.term.Literal(u'bar')),
     (rdflib.term.URIRef(u'http://www.w3.org/2000/01/rdf-schema#label'),
      rdflib.term.Literal(u'foo'))]
    >>> g.add([u, SKOS.prefLabel, Literal('bla')])
    >>> pprint(g.preferredLabel(u))
    [(rdflib.term.URIRef(u'http://www.w3.org/2004/02/skos/core#prefLabel'),
      rdflib.term.Literal(u'bla'))]
    >>> g.add([u, SKOS.prefLabel, Literal('blubb', lang='en')])
    >>> sorted(g.preferredLabel(u))
    [(rdflib.term.URIRef(u'http://www.w3.org/2004/02/skos/core#prefLabel'),
      rdflib.term.Literal(u'bla')),
      (rdflib.term.URIRef(u'http://www.w3.org/2004/02/skos/core#prefLabel'),
      rdflib.term.Literal(u'blubb', lang='en'))]
    >>> g.preferredLabel(u, lang='')
    [(rdflib.term.URIRef(u'http://www.w3.org/2004/02/skos/core#prefLabel'),
      rdflib.term.Literal(u'bla'))]
    >>> pprint(g.preferredLabel(u, lang='en'))
    [(rdflib.term.URIRef(u'http://www.w3.org/2004/02/skos/core#prefLabel'),
      rdflib.term.Literal(u'blubb', lang='en'))]
```

`Graph.`**`triples`**(*(s, p, o)*)

Generator over the triple store

Returns triples that match the given triple pattern. If triple pattern does not provide a context, all contexts will be searched.

`Graph.value`(*subject=None*, *predicate=rdflib.term.URIRef(u'http://www.w3.org/1999/02/22-rdf-syntax-ns#value')*, *object=None*, *default=None*, *any=True*)
Get a value for a pair of two criteria

Exactly one of subject, predicate, object must be None. Useful if one knows that there may only be one value.

It is one of those situations that occur a lot, hence this 'macro' like utility

Parameters: subject, predicate, object – exactly one must be None default – value to be returned if no values found any – if True, return any value in the case there is more than one, else, raise UniquenessError

`Graph.subjects`(*predicate=None*, *object=None*)
A generator of subjects with the given predicate and object

`Graph.objects`(*subject=None*, *predicate=None*)
A generator of objects with the given subject and predicate

`Graph.predicates`(*subject=None*, *object=None*)
A generator of predicates with the given subject and object

`Graph.subject_objects`(*predicate=None*)
A generator of (subject, object) tuples for the given predicate

`Graph.subject_predicates`(*object=None*)
A generator of (subject, predicate) tuples for the given object

`Graph.predicate_objects`(*subject=None*)
A generator of (predicate, object) tuples for the given subject

Querying with SPARQL

Run a Query

The RDFLib comes with an implementation of the SPARQL 1.1 Query and SPARQL 1.1 Update languages.

Queries can be evaluated against a graph with the *rdflib.graph.Graph.query()* method, and updates with *rdflib.graph.Graph.update()*.

The query method returns a *rdflib.query.Result* instance. For SELECT queries, iterating over this return *rdflib.query.ResultRow* instances, each containing a set of variable bindings. For CONSTRUCT/DESCRIBE queries, iterating over the result object gives the triples. For ASK queries, iterating will yield the single boolean answer, or evaluating the result object in a boolean-context (i.e. `bool(result)`)

Continuing the example...

```
import rdflib

g = rdflib.Graph()

# ... add some triples to g somehow ...
g.parse("some_foaf_file.rdf")

qres = g.query(
    """SELECT DISTINCT ?aname ?bname
       WHERE {
          ?a foaf:knows ?b .
          ?a foaf:name ?aname .
```

```
          ?b foaf:name ?bname .
      }""")

for row in qres:
    print("%s knows %s" % row)
```

The results are tuples of values in the same order as your SELECT arguments. Alternatively, the values can be accessed by variable name, either as attributes, or as items: `row.b` and `row["b"]` is equivalent.

```
Timothy Berners-Lee knows Edd Dumbill
Timothy Berners-Lee knows Jennifer Golbeck
Timothy Berners-Lee knows Nicholas Gibbins
Timothy Berners-Lee knows Nigel Shadbolt
Dan Brickley knows binzac
Timothy Berners-Lee knows Eric Miller
Drew Perttula knows David McClosky
Timothy Berners-Lee knows Dan Connolly
...
```

As an alternative to using `PREFIX` in the SPARQL query, namespace bindings can be passed in with the `initNs` kwarg, see `namespace_and_bindings`.

Variables can also be pre-bound, using `initBindings` kwarg can be used to pass in a `dict` of initial bindings, this is particularly useful for prepared queries, as described below.

Prepared Queries

RDFLib lets you *prepare* queries before execution, this saves re-parsing and translating the query into SPARQL Algebra each time.

The method `rdflib.plugins.sparql.prepareQuery()` takes a query as a string and will return a *rdflib.plugins.sparql.sparql.Query* object. This can then be passed to the *rdflib.graph.Graph.query()* method.

The `initBindings` kwarg can be used to pass in a `dict` of initial bindings:

```
q = prepareQuery(
        'SELECT ?s WHERE { ?person foaf:knows ?s .}',
        initNs = { "foaf": FOAF })

g = rdflib.Graph()
g.load("foaf.rdf")

tim = rdflib.URIRef("http://www.w3.org/People/Berners-Lee/card#i")

for row in g.query(q, initBindings={'person': tim}):
        print row
```

Custom Evaluation Functions

For experts, it is possible to override how bits of SPARQL algebra are evaluated. By using the setuptools entry-point `rdf.plugins.sparqleval`, or simply adding to an entry to *rdflib.plugins.sparql.CUSTOM_EVALS*, a custom function can be registered. The function will be called for each algebra component and may raise `NotImplementedError` to indicate that this part should be handled by the default implementation.

See `examples/custom_eval.py`

Utilities and convenience functions

For RDF programming, RDFLib and Python may not execute the fastest, but we try hard to make it the fastest and most convenient way to write!

This is a collection of hints and pointers for hassle free RDF-coding.

User-friendly labels

Use *label()* to quickly look up the RDFS label of something, or better use *preferredLabel()* to find a label using several different properties (i.e. either `rdfs:label`, `skos:preferredLabel`, `dc:title`, etc.).

Functional properties

Use *value()* and *set()* to work with *functional properties*, i.e. properties than can only occur once for a resource.

Slicing graphs

Python allows slicing arrays with a `slice` object, a triple of `start`, `stop` index and step-size:

```
>>> range(10)[2:9:3]
[2, 5, 8]
```

RDFLib graphs override `__getitem__` and we pervert the slice triple to be a RDF triple instead. This lets slice syntax be a shortcut for *triples()*, *subject_predicates()*, `contains()`, and other Graph query-methods:

```
graph[:]
# same as
iter(graph)

graph[bob]
# same as
graph.predicate_objects(bob)

graph[bob : FOAF.knows]
# same as
graph.objects(bob, FOAF.knows)

graph[bob : FOAF.knows : bill]
# same as
(bob, FOAF.knows, bill) in graph

graph[:FOAF.knows]
# same as
graph.subject_objects(FOAF.knows)

...
```

See *examples.slice* for a complete example.

Note: Slicing is convenient for run-once scripts of playing around in the Python `REPL`. However, since slicing returns tuples of varying length depending on which parts of the slice are bound, you should be careful using it in

more complicated programs. If you pass in variables, and they are `None` or `False`, you may suddenly get a generator of different length tuples back than you expect.

SPARQL Paths

SPARQL property paths are possible using overridden operators on URIRefs. See *examples.foafpaths* and *rdflib.paths*.

Serializing a single term to N3

For simple output, or simple serialisation, you often want a nice readable representation of a term. All terms have a `.n3(namespace_manager = None)` method, which will return a suitable N3 format:

```
>>> from rdflib import Graph, URIRef, Literal, BNode
>>> from rdflib.namespace import FOAF, NamespaceManager

>>> person = URIRef('http://xmlns.com/foaf/0.1/Person')
>>> person.n3()
u'<http://xmlns.com/foaf/0.1/Person>'

>>> g = Graph()
>>> g.bind("foaf", FOAF)

>>> person.n3(g.namespace_manager)
u'foaf:Person'

>>> l = Literal(2)
>>> l.n3()
u'"2"^^<http://www.w3.org/2001/XMLSchema#integer>'

>>> l.n3(g.namespace_manager)
u'"2"^^xsd:integer'
```

Parsing data from a string

You can parse data from a string with the `data` param:

```
graph.parse(data = '<urn:a> <urn:p> <urn:b>.', format='n3')
```

Commandline-tools

RDFLib includes a handful of commandline tools, see *rdflib.tools*.

examples Package

These examples all live in `./examples` in the source-distribution of RDFLib.

conjunctive_graphs Module

An RDFLib ConjunctiveGraph is an (unamed) aggregation of all the named graphs within a Store. The *get_context()* method can be used to get a particular named graph, or triples can be added to the default graph

This example shows how to create some named graphs and work with the conjunction of all the graphs.

custom_datatype Module

RDFLib can map between data-typed literals and python objects.

Mapping for integers, floats, dateTimes, etc. are already added, but you can also add your own.

This example shows how *rdflib.term.bind()* lets you register new mappings between literal datatypes and python objects

custom_eval Module

This example shows how a custom evaluation function can be added to handle certain SPARQL Algebra elements

A custom function is added that adds `rdfs:subClassOf` "inference" when asking for `rdf:type` triples.

Here the custom eval function is added manually, normally you would use setuptools and entry_points to do it: i.e. in your setup.py:

```
entry_points = {
    'rdf.plugins.sparqleval': [
        'myfunc =     mypackage:MyFunction',
        ],
}
```

examples.custom_eval.**customEval**(*ctx*, *part*)
Rewrite triple patterns to get super-classes

film Module

film.py: a simple tool to manage your movies review Simon Rozet, http://atonie.org/

@@ : - manage directors and writers - manage actors - handle non IMDB uri - markdown support in comment

Requires download and import of Python imdb library from http://imdbpy.sourceforge.net/ - (warning: installation will trigger automatic installation of several other packages)

– Usage:

film.py whoami "John Doe <john@doe.org>" Initialize the store and set your name and email.

film.py whoami Tell you who you are

film.py http://www.imdb.com/title/tt0105236/ Review the movie "Reservoir Dogs"

class examples.film.**Store**

__init__()

__module__ = 'examples.film'

movie_is_in(*uri*)

new_movie(*movie*)

new_review(*movie*, *date*, *rating*, *comment=None*)

save()

who(*who=None*)

examples.film.**help**()

examples.film.**main**(*argv=None*)

foafpaths Module

SPARQL 1.1 defines path operators for combining/repeating predicates in triple-patterns.

We overload some python operators on URIRefs to allow creating path operators directly in python.

Operator	Path
`p1 / p2`	Path sequence
`p1 | p2`	Path alternative
`p1 * '*'`	chain of 0 or more p's
`p1 * '+'`	chain of 1 or more p's
`p1 * '?'`	0 or 1 p
`~p1`	p1 inverted, i.e. (s p1 o) <=> (o ~p1 s)
`-p1`	NOT p1, i.e. any property but p1

these can then be used in property position for `s,p,o` triple queries for any graph method.

See the docs for *rdflib.paths* for the details.

This example shows how to get the name of friends with a single query.

prepared_query Module

SPARQL Queries be prepared (i.e parsed and translated to SPARQL algebra) by the `rdflib.plugins.sparql.prepareQuery()` method.

When executing, variables can be bound with the `initBindings` keyword parameter

resource Module

RDFLib has a *Resource* class, for a resource-centric API.

A resource acts like a URIRef with an associated graph, and allows quickly adding or querying for triples where this resource is the subject.

rdfa_example Module

A simple example showing how to process RDFa from the web

simple_example Module

sleepycat_example Module

A simple example showing how to use a Sleepycat store to do on-disk persistence.

slice Module

RDFLib Graphs (and Resources) can be "sliced" with [] syntax

This is a short-hand for iterating over triples

Combined with SPARQL paths (see `foafpaths.py`) - quite complex queries can be realised.

See *rdflib.graph.Graph.__getitem__()* for details

smushing Module

A FOAF smushing example.

Filter a graph by normalizing all `foaf:Persons` into URIs based on their `mbox_sha1sum`.

Suppose I got two FOAF documents each talking about the same person (according to `mbox_sha1sum`) but they each used a *rdflib.term.BNode* for the subject. For this demo I've combined those two documents into one file:

This filters a graph by changing every subject with a `foaf:mbox_sha1sum` into a new subject whose URI is based on the `sha1sum`. This new graph might be easier to do some operations on.

An advantage of this approach over other methods for collapsing BNodes is that I can incrementally process new FOAF documents as they come in without having to access my ever-growing archive. Even if another `65b983bb397fb71849da910996741752ace8369b` document comes in next year, I would still give it the same stable subject URI that merges with my existing data.

sparql_query_example Module

SPARQL Query using *rdflib.graph.Graph.query()*

The method returns a *Result*, iterating over this yields *ResultRow* objects

The variable bindings can be access as attributes of the row objects For variable names that are not valid python identifiers, dict access (i.e. with `row[var]` / `__getitem__`) is also possible.

`vars` contains the variables

sparql_update_example Module

SPARQL Update statements can be applied with *rdflib.graph.Graph.update()*

sparqlstore_example Module

A simple example showing how to use the SPARQLStore

swap_primer Module

This is a simple primer using some of the example stuff in the Primer on N3:

http://www.w3.org/2000/10/swap/Primer

transitive Module

An example illustrating how to use the *transitive_subjects()* and *transitive_objects()* graph methods

Formal definition

The *transitive_objects()* method finds all nodes such that there is a path from subject to one of those nodes using only the predicate property in the triples. The *transitive_subjects()* method is similar; it finds all nodes such that there is a path from the node to the object using only the predicate property.

Informal description, with an example

In brief, *transitive_objects()* walks forward in a graph using a particular property, and *transitive_subjects()* walks backward. A good example uses a property `ex:parent`, the semantics of which are biological parentage. The *transitive_objects()* method would get all the ancestors of a particular person (all nodes such that there is a parent path between the person and the object). The *transitive_subjects()* method would get all the descendants of a particular person (all nodes such that there is a parent path between the node and the person). So, say that your URI is `ex:person`.

This example would get all of your (known) ancestors, and then get all the (known) descendants of your maternal grandmother.

> **Warning:** The `transitive_objects()` method has the start node as the *first* argument, but the `transitive_subjects()` method has the start node as the *second* argument.

User-defined transitive closures

The method *transitiveClosure()* returns transtive closures of user-defined functions.

CHAPTER 2

In depth

If you already worked with RDF and need to know the peculiarities of RDFLib, these are for you.

RDF terms in rdflib

Terms are the kinds of objects that can appear in a quoted/asserted triples. Those that are part of core RDF concepts are: `Blank Node`, `URI Reference` and `Literal`, the latter consisting of a literal value and either a datatype or an **RFC 3066** language tag.

All terms in RDFLib are sub-classes of the *`rdflib.term.Identifier`* class.

Nodes are a subset of the Terms that the underlying store actually persists. The set of such Terms depends on whether or not the store is formula-aware. Stores that aren't formula-aware would only persist those terms core to the RDF Model, and those that are formula-aware would be able to persist the N3 extensions as well. However, utility terms that only serve the purpose for matching nodes by term-patterns probably will only be terms and not nodes.

BNodes

> In RDF, a blank node (also called BNode) is a node in an RDF graph representing a resource for which a URI or literal is not given. The resource represented by a blank node is also called an anonymous resource. By RDF standard a blank node can only be used as subject or object in an RDF triple, although in some syntaxes like Notation 3 [1] it is acceptable to use a blank node as a predicate. If a blank node has a node ID (not all blank nodes are labelled in all RDF serializations), it is limited in scope to a serialization of a particular RDF graph, i.e. the node p1 in the subsequent example does not represent the same node as a node named p1 in any other graph –wikipedia

class `rdflib.term.`**`BNode`**

Blank Node: http://www.w3.org/TR/rdf-concepts/#section-blank-nodes

```
>>> from rdflib import BNode
>>> anode = BNode()
>>> anode
rdflib.term.BNode('AFwALAKU0')
>>> anode.n3()
u'_:AFwALAKU0'
```

URIRefs

A URI reference within an RDF graph is a Unicode string that does not contain any control characters (#x00 - #x1F, #x7F-#x9F) and would produce a valid URI character sequence representing an absolute URI with optional fragment identifier – W3 RDF Concepts

class rdflib.term.**URIRef**
RDF URI Reference: http://www.w3.org/TR/rdf-concepts/#section-Graph-URIref

```
>>> from rdflib import URIRef
>>> aref = URIRef()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: __new__() takes at least 2 arguments (1 given)
>>> aref = URIRef('')
>>> aref
rdflib.term.URIRef(u'')
>>> aref = URIRef('http://example.com')
>>> aref
rdflib.term.URIRef(u'http://example.com')
>>> aref.n3()
u'<http://example.com>'
```

Literals

Literals are the attribute values in RDF, for instance, a person's name, the date of birth, height, etc. Literals can have a data-type (i.e. this is a double) or a language tag (this label is in English).

class rdflib.term.**Literal**
RDF Literal: http://www.w3.org/TR/rdf-concepts/#section-Graph-Literal

The lexical value of the literal is the unicode object The interpreted, datatyped value is available from .value

Language tags must be valid according to :rfc:5646

For valid XSD datatypes, the lexical form is optionally normalized at construction time. Default behaviour is set by rdflib.NORMALIZE_LITERALS and can be overridden by the normalize parameter to __new__

Equality and hashing of Literals are done based on the lexical form, i.e.:

```
>>> from rdflib.namespace import XSD
```

```
>>> Literal('01')!=Literal('1') # clear - strings differ
True
```

but with data-type they get normalized:

```
>>> Literal('01', datatype=XSD.integer)!=Literal('1', datatype=XSD.integer)
False
```

unless disabled:

```
>>> Literal('01', datatype=XSD.integer, normalize=False)!=Literal('1', datatype=XSD.integer)
True
```

Value based comparison is possible:

```
>>> Literal('01', datatype=XSD.integer).eq(Literal('1', datatype=XSD.float))
True
```

The eq method also provides limited support for basic python types:

```
>>> Literal(1).eq(1) # fine - int compatible with xsd:integer
True
>>> Literal('a').eq('b') # fine - str compatible with plain-lit
False
>>> Literal('a', datatype=XSD.string).eq('a') # fine - str compatible with xsd:string
True
>>> Literal('a').eq(1) # not fine, int incompatible with plain-lit
NotImplemented
```

Greater-than/less-than ordering comparisons are also done in value space, when compatible datatypes are used. Incompatible datatypes are ordered by DT, or by lang-tag. For other nodes the ordering is None < BNode < URIRef < Literal

Any comparison with non-rdflib Node are "NotImplemented" In PY2.X some stable order will be made up by python

In PY3 this is an error.

```
>>> from rdflib import Literal, XSD
>>> lit2006 = Literal('2006-01-01',datatype=XSD.date)
>>> lit2006.toPython()
datetime.date(2006, 1, 1)
>>> lit2006 < Literal('2007-01-01',datatype=XSD.date)
True
>>> Literal(datetime.utcnow()).datatype
rdflib.term.URIRef(u'http://www.w3.org/2001/XMLSchema#dateTime')
>>> Literal(1) > Literal(2) # by value
False
>>> Literal(1) > Literal(2.0) # by value
False
>>> Literal('1') > Literal(1) # by DT
True
>>> Literal('1') < Literal('1') # by lexical form
False
>>> Literal('a', lang='en') > Literal('a', lang='fr') # by lang-tag
False
>>> Literal(1) > URIRef('foo') # by node-type
True
```

The > < operators will eat this NotImplemented and either make up an ordering (py2.x) or throw a TypeError (py3k):

```
>>> Literal(1).__gt__(2.0)
NotImplemented
```

A literal in an RDF graph contains one or two named components.

All literals have a lexical form being a Unicode string, which SHOULD be in Normal Form C.

Plain literals have a lexical form and optionally a language tag as defined by **RFC 3066**, normalized to lowercase. An exception will be raised if illegal language-tags are passed to `rdflib.term.Literal.__init__()`.

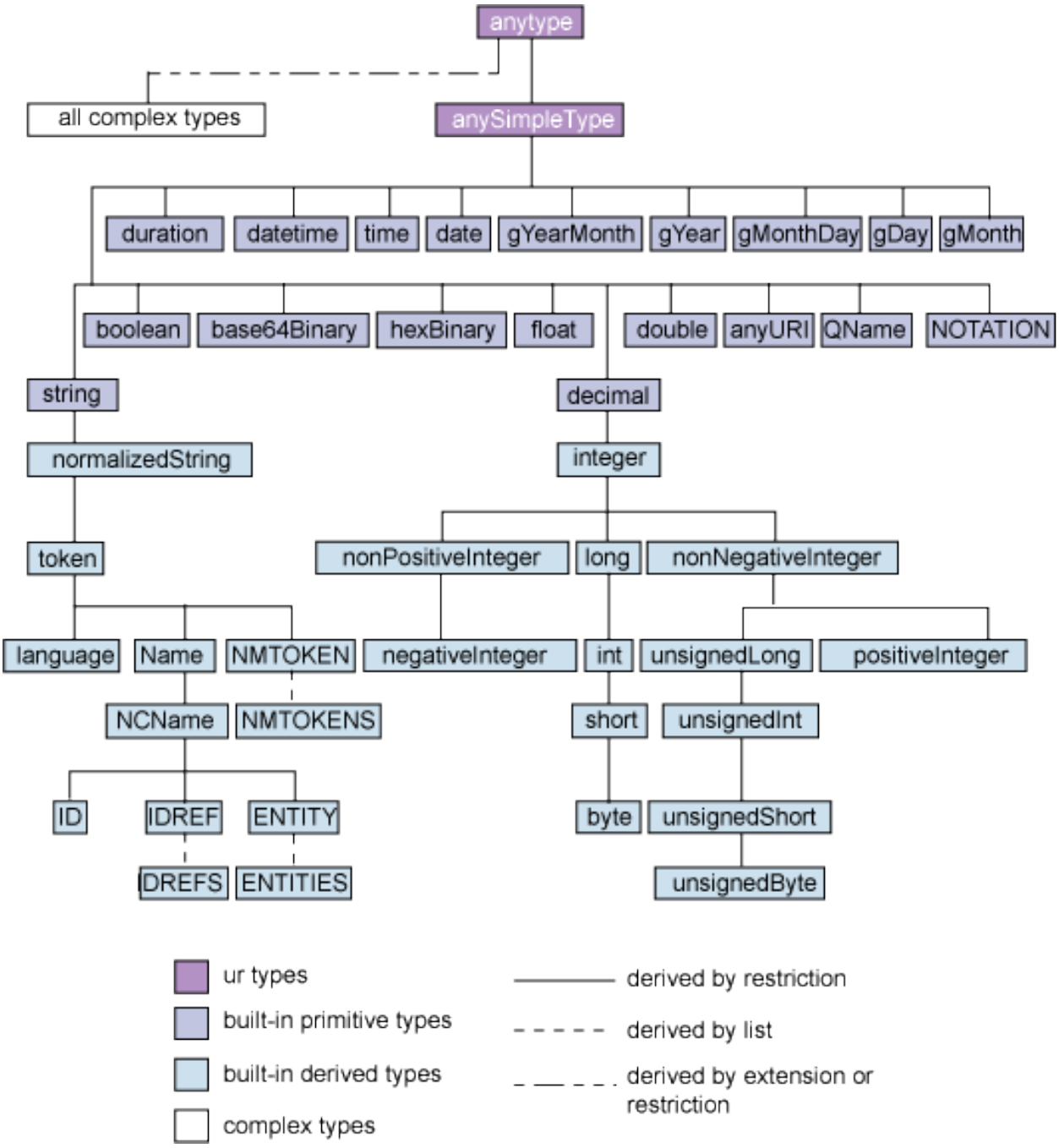
Typed literals have a lexical form and a datatype URI being an RDF URI reference.

Note: When using the language tag, care must be taken not to confuse language with locale. The language tag relates only to human language text. Presentational issues should be addressed in end-user applications.

Note: The case normalization of language tags is part of the description of the abstract syntax, and consequently the abstract behaviour of RDF applications. It does not constrain an RDF implementation to actually normalize the case. Crucially, the result of comparing two language tags should not be sensitive to the case of the original input. – RDF Concepts and Abstract Syntax

Python support

RDFLib Literals essentially behave like unicode characters with an XML Schema datatype or language attribute.



The class provides a mechanism to both convert Python literals (and their built-ins such as time/date/datetime) into equivalent RDF Literals and (conversely) convert Literals to their Python equivalent. This mapping to and from Python literals is done as follows:

XML Datatype	Python type
None	None [1]
xsd:time	time [2]
xsd:date	date
xsd:dateTime	datetime
xsd:string	None
xsd:normalizedString	None
xsd:token	None
xsd:language	None
xsd:boolean	boolean
xsd:decimal	Decimal
xsd:integer	long
xsd:nonPositiveInteger	int
xsd:long	long
xsd:nonNegativeInteger	int
xsd:negativeInteger	int
xsd:int	long
xsd:unsignedLong	long
xsd:positiveInteger	int
xsd:short	int
xsd:unsignedInt	long
xsd:byte	int
xsd:unsignedShort	int
xsd:unsignedByte	int
xsd:float	float
xsd:double	float
xsd:base64Binary	`base64`
xsd:anyURI	None
rdf:XMLLiteral	`xml.dom.minidom.Document` [3]
rdf:HTML	`xml.dom.minidom.DocumentFragment`

An appropriate data-type and lexical representation can be found using:

`rdflib.term.`**`_castPythonToLiteral`**(*obj*)
Casts a python datatype to a tuple of the lexical value and a datatype URI (or None)

and the other direction with

`rdflib.term.`**`_castLexicalToPython`**(*lexical*, *datatype*)
Map a lexical form to the value-space for the given datatype :returns: a python object for the value or `None`

All this happens automatically when creating `Literal` objects by passing Python objects to the constructor, and you never have to do this manually.

You can add custom data-types with *`rdflib.term.bind()`*, see also *`examples.custom_datatype`*

[1] plain literals map directly to value space
[2] Date, time and datetime literals are mapped to Python instances using the isodate package).
[3] this is a bit dirty - by accident the `html5lib` parser produces `DocumentFragments`, and the xml parser `Documents`, letting us use this to decide what datatype when round-tripping.

Namespaces and Bindings

RDFLib provides several short-cuts to working with many URIs in the same namespace.

The *rdflib.namespace* defines the *rdflib.namespace.Namespace* class which lets you easily create URIs in a namespace:

```
from rdflib import Namespace

n = Namespace("http://example.org/")
n.Person # as attribute
# = rdflib.term.URIRef(u'http://example.org/Person')

n['first%20name'] # as item - for things that are not valid python identifiers
# = rdflib.term.URIRef(u'http://example.org/first%20name')
```

The `namespace` module also defines many common namespaces such as RDF, RDFS, OWL, FOAF, SKOS, etc.

Namespaces can also be associated with prefixes, in a *rdflib.namespace.NamespaceManager*, i.e. using `foaf` for `http://xmlns.com/foaf/0.1/`. Each RDFLib graph has a *namespace_manager* that keeps a list of namespace to prefix mappings. The namespace manager is populated when reading in RDF, and these prefixes are used when serialising RDF, or when parsing SPARQL queries. Additional prefixes can be bound with the `rdflib.graph.bind()` method.

Namespaces in SPARQL Queries

The `initNs` argument supplied to *query()* is a dictionary of namespaces to be expanded in the query string. If you pass no `initNs` argument, the namespaces registered with the graphs namespace_manager are used:

```
...
from rdflib.namespace import FOAF
graph.query('SELECT * WHERE { ?p a foaf:Person }', initNs={ 'foaf': FOAF })
```

In order to use an empty prefix (e.g. `?a :knows ?b`), use a `PREFIX` directive with no prefix in the SPARQL query to set a default namespace:

```
PREFIX : <http://xmlns.com/foaf/0.1/>
```

Persistence

RDFLib provides an *abstracted Store API* for persistence of RDF and Notation 3. The *Graph* class works with instances of this API (as the first argument to its constructor) for triple-based management of an RDF store including: garbage collection, transaction management, update, pattern matching, removal, length, and database management (*open()* / *close()* / *destroy()*).

Additional persistence mechanisms can be supported by implementing this API for a different store.

Stores currently shipped with core RDFLib

- *Memory* (not persistent!)
- *Sleepycat* (on disk persistence via Python's bsddb or bsddb3 packages)
- *SPARQLStore* - a read-only wrapper around a remote SPARQL Query endpoint.
- *SPARQLUpdateStore* - a read-write wrapper around a remote SPARQL query/update endpoint pair.

Usage

Most cases passing the name of the store to the Graph constructor is enough:

```
from rdflib import Graph

graph = Graph(store='Sleepycat')
```

Most store offering on-disk persistence will need to be opened before reading or writing :

```
graph = Graph('Sleepycat')

# first time create the store:
graph.open('/home/user/data/myRDFLibStore', create = True)

# work with the graph:
graph.add( mytriples )

# when done!
graph.close()
```

When done, `close()` must be called to free the resources associated with the store.

Additional store plugins

More store implementations are available in RDFLib extension projects:

- rdflib-sqlalchemy, which supports stored on a wide-variety of RDBMs backends,
- rdflib-leveldb - a store on to of Google's LevelDB key-value store.
- rdflib-kyotocabinet - a store on to of the Kyoto Cabinet key-value store.

Example

- `examples.sleepycat_example` contains an example for using a Sleepycat store.
- `examples.sparqlstore_example` contains an example for using a SPARQLStore.

Merging graphs

A merge of a set of RDF graphs is defined as follows. If the graphs in the set have no blank nodes in common, then the union of the graphs is a merge; if they do share blank nodes, then it is the union of a set of graphs that is obtained by replacing the graphs in the set by equivalent graphs that share no blank nodes. This is often described by saying that the blank nodes have been 'standardized apart'. It is easy to see that any two merges are equivalent, so we will refer to the merge, following the convention on equivalent graphs. Using the convention on equivalent graphs and identity, any graph in the original set is considered to be a subgraph of the merge.

One does not, in general, obtain the merge of a set of graphs by concatenating their corresponding N-Triples documents and constructing the graph described by the merged document. If some of the documents use the same node identifiers, the merged document will describe a graph in which some of the blank nodes have been 'accidentally' identified. To merge N-Triples documents it is necessary to check if the same nodeID is used in two or more documents, and to replace it with a distinct nodeID in each of

them, before merging the documents. Similar cautions apply to merging graphs described by RDF/XML documents which contain nodeIDs

(copied directly from http://www.w3.org/TR/rdf-mt/#graphdefs)

In RDFLib, blank nodes are given unique IDs when parsing, so graph merging can be done by simply reading several files into the same graph:

```
from rdflib import Graph

graph = Graph()

graph.parse(input1)
graph.parse(input2)
```

`graph` now contains the merged graph of `input1` and `input2`.

Note: However, the set-theoretic graph operations in RDFLib are assumed to be performed in sub-graphs of some larger data-base (for instance, in the context of a *`ConjunctiveGraph`*) and assume shared blank node IDs, and therefore do NOT do *correct* merging, i.e.:

```
from rdflib import Graph

g1 = Graph()
g1.parse(input1)

g2 = Graph()
g2.parse(input2)

graph = g1 + g2
```

May cause unwanted collisions of blank-nodes in `graph`.

Upgrading from RDFLib version 3.X to 4.X

RDFLib version 4.0 introduced a small number of backwards compatible changes that you should know about when porting code from version 3 or earlier.

SPARQL and SPARQLStore are now included in core

For version 4.0 we've merged the SPARQL implementation from `rdflib-sparql`, the SPARQL(Update)Store from `rdflib-sparqlstore` and miscellaneous utilities from `rdfextras`. If you used any of these previously, everything you need should now be included.

Datatyped literals

We separate lexical and value space operations for datatyped literals.

This mainly affects the way datatyped literals are compared. Lexical space comparisons are done by default for == and !=, meaning the exact lexical representation and the exact data-types have to match for literals to be equal. Value space comparisons are also available through the *`rdflib.term.Identifier.eq()`* and *`rdflib.term.Identifier.neq()`* methods, < > <= >= are also done in value space.

Most things now work in a fairly sane and sensible way, if you do not have existing stores/intermediate stored sorted lists, or hash-dependent something-or-other, you should be good to go.

Things to watch out for:

Literals no longer compare equal across data-types with `'=='`

i.e.

```
>>> Literal(2, datatype=XSD.int) == Literal(2, datatype=XSD.float)
False
```

But a new method *rdflib.term.Identifier.eq()* on all Nodes has been introduced, which does semantic equality checking, i.e.:

```
>>> Literal(2, datatype=XSD.int).eq(Literal(2, datatype=XSD.float))
True
```

The `eq` method is still limited to what data-types map to the same *value space*, i.e. all numeric types map to numbers and will compare, `xsd:string` and `plain literals` both map to strings and compare fine, but:

```
>>> Literal(2, datatype=XSD.int).eq(Literal('2'))
False
```

Literals will be normalised according to datatype

If you care about the exact lexical representation of a literal, and not just the value. Either set `rdflib.NORMALIZE_LITERALS` to `False` before creating your literal, or pass `normalize=False` to the Literal constructor

Ordering of literals and nodes has changed

Comparing literals with `<, >, <=, >=` now work same as in SPARQL filter expressions.

Greater-than/less-than ordering comparisons are also done in value space, when compatible datatypes are used. Incompatible datatypes are ordered by data-type, or by lang-tag. For other nodes the ordering is `None < BNode < URIRef < Literal`

Any comparison with non-rdflib Node are `NotImplemented` In PY2.X some stable order will be made up by python. In PY3 this is an error.

Custom mapping of datatypes to python objects

You can add new mappings of datatype URIs to python objects using the *rdflib.term.bind()* method. This also allows you to specify a constructor for constructing objects from the lexical string representation, and a serialization method for generating a lexical string representation from an object.

Minor Changes

- *rdflib.namespace.Namespace* is no longer a sub-class of *rdflib.term.URIRef* this was changed as it makes no sense for a namespace to be a node in a graph, and was causing numerous bug. Unless you do something very special, you should not notice this change.

- The identifiers for Graphs are now converted to URIRefs if they are not a *rdflib.term.Node*, i.e. no more graphs with string identifiers. Again, unless you do something quite unusual, you should not notice.
- String concatenation with URIRefs now returns URIRefs, not strings:

```
>>> URIRef("http://example.org/people/") + "Bill"
```

rdflib.term.URIRef(u'http://example.org/people/Bill')

This is be convenient, but may cause trouble if you expected a string.

Upgrading from RDFLib version 2.X to 3.X

Introduction

This page details the changes required to upgrade from RDFLib 2.X to 3.X.

Some older Linux distributions still ship 2.4.X. If needed, you can also install 2.4 using easy_install/setup tools.

Version 3.0 reorganised some packages, and moved non-core parts of rdflib to the rdfextras project

Features moved to rdfextras

- SPARQL Support is now in rdfextras / rdflib-sparql
- The RDF Commandline tools are now in rdfextras

Warning: If you install packages with just distutils - you will need to register the sparql plugins manually - we strongly recommend installing with setuptools or distribute! To register the plugins add this somewhere in your program:

```
rdflib.plugin.register('sparql', rdflib.query.Processor,
                       'rdfextras.sparql.processor', 'Processor')
rdflib.plugin.register('sparql', rdflib.query.Result,
                       'rdfextras.sparql.query', 'SPARQLQueryResult')
```

Unstable features that were removed

The RDBMS back stores (MySQL/PostgreSQL) were removed, but are in the process of being moved to rdfextras. The Redland, SQLite and ZODB stores were all removed.

Packages/Classes that were renamed

Previously all packages and classes had colliding names, i.e. both package and the class was called "Graph":

```
from rdflib.Graph import Graph, ConjunctiveGraph
```

Now all packages are lower-case, i.e:

```
from rdflib.graph import Graph, ConjunctiveGraph
```

Most classes you need are available from the top level rdflib package:

```
from rdflib import Graph, URIRef, BNode, Literal
```

Namespace classes for RDF, RDFS, OWL are now directly in the rdflib package, i.e. in 2.4:

```
from rdflib.RDF import RDFNS as RDF
```

in 3.0:

```
from rdflib import RDF
```

Frequently Asked Questions about using RDFLib

Questions about parsing

Questions about manipulating

Questions about serializing

Which serialization method is the most efficient?

Currently, the "nt" output format uses the most efficient serialization; "rdf/xml" should also be efficient. You can serialize to these formats using code similar to the following:

```
myGraph.serialize(target_nt, format="nt")
myGraph.serialize(target_rdfxml, format="xml")
```

How can I use some of the abbreviated RDF/XML syntax?

Use the "pretty-xml" *format* argument to the *serialize* method:

```
myGraph.serialize(target_pretty, format="pretty-xml")
```

How can I control the binding of prefixes to XML namespaces when using RDF/XML?

Each graph comes with a NamespaceManager instance in the *namespace_manager* field; you can use the *bind* method of this instance to bind a prefix to a namespace URI:

```
myGraph.namespace_manager.bind('prefix', URIRef('scheme:my-namespace-uri:'))
```

Does RDFLib support serialization to the TriX format?

Yes, both parsing and serialising is supported:

```
graph.serialize(format="trix") and graph.load(source, format="trix")
```

CHAPTER 3

Reference

The nitty-gritty details of everything.

Plugins

Many parts of RDFLib are extensible with plugins through setuptools entry-points. These pages list the plugins included in RDFLib core.

Plugin parsers

These serializers are available in default RDFLib, you can use them by passing the name to graph's *parse()* method:

```
graph.parse(my_url, format='n3')
```

The `html` parser will auto-detect RDFa, HTurtle or Microdata.

It is also possible to pass a mime-type for the `format` parameter:

```
graph.parse(my_url, format='application/rdf+xml')
```

If you are not sure what format your file will be, you can use *rdflib.util.guess_format()* which will guess based on the file extension.

Name	Class
html	*StructuredDataParser*
hturtle	*HTurtleParser*
mdata	*MicrodataParser*
microdata	*MicrodataParser*
n3	*N3Parser*
nquads	*NQuadsParser*
nt	*NTParser*
rdfa	*RDFaParser*
rdfa1.0	*RDFa10Parser*
rdfa1.1	*RDFaParser*
trix	*TriXParser*
turtle	*TurtleParser*
xml	*RDFXMLParser*

Plugin serializers

These serializers are available in default RDFLib, you can use them by passing the name to a graph's *serialize()* method:

```
print graph.serialize(format='n3')
```

It is also possible to pass a mime-type for the `format` parameter:

```
graph.serialize(my_url, format='application/rdf+xml')
```

Name	Class
n3	*N3Serializer*
nquads	*NQuadsSerializer*
nt	*NTSerializer*
pretty-xml	*PrettyXMLSerializer*
trig	*TrigSerializer*
trix	*TriXSerializer*
turtle	*TurtleSerializer*
xml	*XMLSerializer*

Plugin stores

Name	Class
Auditable	*AuditableStore*
Concurrent	*ConcurrentStore*
IOMemory	*IOMemory*
SPARQLStore	*SPARQLStore*
SPARQLUpdateStore	*SPARQLUpdateStore*
Sleepycat	*Sleepycat*
default	*IOMemory*

Plugin query results

Plugins for reading and writing of (SPARQL) `QueryResult` - pass `name` to either `parse()` or `serialize()`

Parsers

Name	Class
csv	*CSVResultParser*
json	*JSONResultParser*
tsv	*TSVResultParser*
xml	*XMLResultParser*

Serializers

Name	Class
csv	*CSVResultSerializer*
json	*JSONResultSerializer*
txt	TXTResultSerializer
xml	*XMLResultSerializer*

rdflib API docs

rdflib Package

rdflib Package

A pure Python package providing the core RDF constructs.

The packages is intended to provide the core RDF types and interfaces for working with RDF. The package defines a plugin interface for parsers, stores, and serializers that other packages can use to implement parsers, stores, and serializers that will plug into the rdflib package.

The primary interface *rdflib* exposes to work with RDF is *rdflib.graph.Graph*.

A tiny example:

```
>>> from rdflib import Graph, URIRef, Literal
```

```
>>> g = Graph()
>>> result = g.parse("http://www.w3.org/2000/10/swap/test/meet/blue.rdf")
```

```
>>> print("graph has %s statements." % len(g))
graph has 4 statements.
>>>
>>> for s, p, o in g:
...     if (s, p, o) not in g:
...         raise Exception("It better be!")
```

```
>>> s = g.serialize(format='nt')
>>>
>>> sorted(g) == [
...  (URIRef(u'http://meetings.example.com/cal#m1'),
...   URIRef(u'http://www.example.org/meeting_organization#homePage'),
...   URIRef(u'http://meetings.example.com/m1/hp')),
...  (URIRef(u'http://www.example.org/people#fred'),
...   URIRef(u'http://www.example.org/meeting_organization#attending'),
...   URIRef(u'http://meetings.example.com/cal#m1')),
...  (URIRef(u'http://www.example.org/people#fred'),
...   URIRef(u'http://www.example.org/personal_details#GivenName'),
...   Literal(u'Fred')),
...  (URIRef(u'http://www.example.org/people#fred'),
...   URIRef(u'http://www.example.org/personal_details#hasEmail'),
...   URIRef(u'mailto:fred@example.com'))
... ]
True
```

rdflib.__init__.**NORMALIZE_LITERALS = True**

If True - Literals lexical forms are normalized when created. I.e. the lexical forms is parsed according to data-type, then the stored lexical form is the re-serialized value that was parsed.

Illegal values for a datatype are simply kept. The normalized keyword for Literal.__new__ can override this.

For example:

```
>>> from rdflib import Literal,XSD
>>> Literal("01", datatype=XSD.int)
rdflib.term.Literal(u'1', datatype=rdflib.term.URIRef(u'http://www.w3.org/2001/XMLSchema#integer
```

This flag may be changed at any time, but will only affect literals created after that time, previously created literals will remain (un)normalized.

rdflib.__init__.**DAWG_LITERAL_COLLATION** = False
DAWG_LITERAL_COLLATION determines how literals are ordered or compared to each other.

In SPARQL, applying the >,<,>=,<= operators to literals of incompatible data-types is an error, i.e:

Literal(2)>Literal('cake') is neither true nor false, but an error.

This is a problem in PY3, where lists of Literals of incompatible types can no longer be sorted.

Setting this flag to True gives you strict DAWG/SPARQL compliance, setting it to False will order Literals with incompatible datatypes by datatype URI

In particular, this determines how the rich comparison operators for Literal work, eq, __neq__, __lt__, etc.

collection Module

class rdflib.collection.**Collection**(*graph*, *uri*, *seq=[]*)
Bases: object

See 3.3.5 Emulating container types: http://docs.python.org/ref/sequence-types.html#l2h-232

```
>>> from rdflib.graph import Graph
>>> from pprint import pprint
>>> listName = BNode()
>>> g = Graph('IOMemory')
>>> listItem1 = BNode()
>>> listItem2 = BNode()
>>> g.add((listName, RDF.first, Literal(1)))
>>> g.add((listName, RDF.rest, listItem1))
>>> g.add((listItem1, RDF.first, Literal(2)))
>>> g.add((listItem1, RDF.rest, listItem2))
>>> g.add((listItem2, RDF.rest, RDF.nil))
>>> g.add((listItem2, RDF.first, Literal(3)))
>>> c = Collection(g,listName)
>>> pprint([term.n3() for term in c])
[u'"1"^^<http://www.w3.org/2001/XMLSchema#integer>',
 u'"2"^^<http://www.w3.org/2001/XMLSchema#integer>',
 u'"3"^^<http://www.w3.org/2001/XMLSchema#integer>']
```

```
>>> Literal(1) in c
True
>>> len(c)
3
>>> c._get_container(1) == listItem1
True
>>> c.index(Literal(2)) == 1
True
```

__delitem__(*key*)

```
>>> from rdflib.namespace import RDF, RDFS
>>> from rdflib import Graph
>>> from pprint import pformat
>>> g = Graph()
>>> a = BNode('foo')
>>> b = BNode('bar')
```

```
        >>> c = BNode('baz')
        >>> g.add((a, RDF.first, RDF.type))
        >>> g.add((a, RDF.rest, b))
        >>> g.add((b, RDF.first, RDFS.label))
        >>> g.add((b, RDF.rest, c))
        >>> g.add((c, RDF.first, RDFS.comment))
        >>> g.add((c, RDF.rest, RDF.nil))
        >>> len(g)
        6
        >>> def listAncestry(node, graph):
        ...   for i in graph.subjects(RDF.rest, node):
        ...     yield i
        >>> [str(node.n3())
        ...   for node in g.transitiveClosure(listAncestry, RDF.nil)]
        ['_:baz', '_:bar', '_:foo']
        >>> lst = Collection(g, a)
        >>> len(lst)
        3
        >>> b == lst._get_container(1)
        True
        >>> c == lst._get_container(2)
        True
        >>> del lst[1]
        >>> len(lst)
        2
        >>> len(g)
        4
```

__getitem__(*key*)
TODO

__init__(*graph*, *uri*, *seq=[]*)

__iter__()
Iterator over items in Collections

__len__()
length of items in collection.

__module__ = 'rdflib.collection'

__setitem__(*key*, *value*)
TODO

append(*item*)

```
        >>> from rdflib.graph import Graph
        >>> listName = BNode()
        >>> g = Graph()
        >>> c = Collection(g,listName,[Literal(1),Literal(2)])
        >>> links = [
        ...     list(g.subjects(object=i, predicate=RDF.first))[0] for i in c]
        >>> len([i for i in links if (i,RDF.rest, RDF.nil) in g])
        1
```

clear()

index(*item*)
Returns the 0-based numerical index of the item in the list

n3()

```
        >>> from rdflib.graph import Graph
        >>> listName = BNode()
        >>> g = Graph('IOMemory')
        >>> listItem1 = BNode()
        >>> listItem2 = BNode()
        >>> g.add((listName, RDF.first, Literal(1)))
        >>> g.add((listName, RDF.rest, listItem1))
        >>> g.add((listItem1, RDF.first, Literal(2)))
        >>> g.add((listItem1, RDF.rest, listItem2))
        >>> g.add((listItem2, RDF.rest, RDF.nil))
        >>> g.add((listItem2, RDF.first, Literal(3)))
        >>> c = Collection(g, listName)
        >>> print(c.n3())
        ( "1"^^<http://www.w3.org/2001/XMLSchema#integer>
          "2"^^<http://www.w3.org/2001/XMLSchema#integer>
          "3"^^<http://www.w3.org/2001/XMLSchema#integer> )
```

compare Module

A collection of utilities for canonicalizing and inspecting graphs.

Among other things, they solve of the problem of deterministic bnode comparisons.

Warning: the time to canonicalize bnodes may increase exponentially on degenerate larger graphs. Use with care!

Example of comparing two graphs:

```
>>> g1 = Graph().parse(format='n3', data='''
...     @prefix : <http://example.org/ns#> .
...     <http://example.org> :rel
...         <http://example.org/same>,
...         [ :label "Same" ],
...         <http://example.org/a>,
...         [ :label "A" ] .
... ''')
>>> g2 = Graph().parse(format='n3', data='''
...     @prefix : <http://example.org/ns#> .
...     <http://example.org> :rel
...         <http://example.org/same>,
...         [ :label "Same" ],
...         <http://example.org/b>,
...         [ :label "B" ] .
... ''')
>>>
>>> iso1 = to_isomorphic(g1)
>>> iso2 = to_isomorphic(g2)
```

These are not isomorphic:

```
>>> iso1 == iso2
False
```

Diff the two graphs:

```
>>> in_both, in_first, in_second = graph_diff(iso1, iso2)
```

Present in both:

```
>>> def dump_nt_sorted(g):
...     for l in sorted(g.serialize(format='nt').splitlines()):
...         if l: print(l.decode('ascii'))

>>> dump_nt_sorted(in_both)
<http://example.org>
    <http://example.org/ns#rel> <http://example.org/same> .
<http://example.org>
    <http://example.org/ns#rel> _:cbcaabaaba17fecbc304a64f8edee4335e .
_:cbcaabaaba17fecbc304a64f8edee4335e
    <http://example.org/ns#label> "Same" .
```

Only in first:

```
>>> dump_nt_sorted(in_first)
<http://example.org>
    <http://example.org/ns#rel> <http://example.org/a> .
<http://example.org>
    <http://example.org/ns#rel> _:cb124e4c6da0579f810c0ffe4eff485bd9 .
_:cb124e4c6da0579f810c0ffe4eff485bd9
    <http://example.org/ns#label> "A" .
```

Only in second:

```
>>> dump_nt_sorted(in_second)
<http://example.org>
    <http://example.org/ns#rel> <http://example.org/b> .
<http://example.org>
    <http://example.org/ns#rel> _:cb558f30e21ddfc05ca53108348338ade8 .
_:cb558f30e21ddfc05ca53108348338ade8
    <http://example.org/ns#label> "B" .
```

class rdflib.compare.**IsomorphicGraph**(***kwargs*)

Bases: *rdflib.graph.ConjunctiveGraph*

An implementation of the RGDA1 graph digest algorithm.

An implementation of RGDA1 (publication forthcoming), a combination of Sayers & Karp's graph digest algorithm using sum and SHA-256 <http://www.hpl.hp.com/techreports/2003/HPL-2003-235R1.pdf> and traces <http://pallini.di.uniroma1.it>, an average case polynomial time algorithm for graph canonicalization.

__eq__(*other*)

Graph isomorphism testing.

__init__(***kwargs*)

__module__ = 'rdflib.compare'

__ne__(*other*)

Negative graph isomorphism testing.

graph_digest(*stats=None*)

Synonym for IsomorphicGraph.internal_hash.

internal_hash(*stats=None*)

This is defined instead of __hash__ to avoid a circular recursion scenario with the Memory store for rdflib which requires a hash lookup in order to return a generator of triples.

rdflib.compare.**to_isomorphic**(*graph*)

rdflib.compare.**isomorphic**(*graph1*, *graph2*)

Compare graph for equality.

Uses an algorithm to compute unique hashes which takes bnodes into account.

Examples:

```
>>> g1 = Graph().parse(format='n3', data='''
...     @prefix : <http://example.org/ns#> .
...     <http://example.org> :rel <http://example.org/a> .
...     <http://example.org> :rel <http://example.org/b> .
...     <http://example.org> :rel [ :label "A bnode." ] .
... ''')
>>> g2 = Graph().parse(format='n3', data='''
...     @prefix ns: <http://example.org/ns#> .
...     <http://example.org> ns:rel [ ns:label "A bnode." ] .
...     <http://example.org> ns:rel <http://example.org/b>,
...             <http://example.org/a> .
... ''')
>>> isomorphic(g1, g2)
True

>>> g3 = Graph().parse(format='n3', data='''
...     @prefix : <http://example.org/ns#> .
...     <http://example.org> :rel <http://example.org/a> .
...     <http://example.org> :rel <http://example.org/b> .
...     <http://example.org> :rel <http://example.org/c> .
... ''')
>>> isomorphic(g1, g3)
False
```

rdflib.compare.**to_canonical_graph**(*g1*)
Creates a canonical, read-only graph.

Creates a canonical, read-only graph where all bnode id:s are based on deterministical SHA-256 checksums, correlated with the graph contents.

rdflib.compare.**graph_diff**(*g1*, *g2*)
Returns three sets of triples: “in both”, “in first” and “in second”.

rdflib.compare.**similar**(*g1*, *g2*)
Checks if the two graphs are “similar”.

Checks if the two graphs are “similar”, by comparing sorted triples where all bnodes have been replaced by a singular mock bnode (the _MOCK_BNODE).

This is a much cheaper, but less reliable, alternative to the comparison algorithm in isomorphic.

compat Module

rdflib.compat.**numeric_greater**(*a*, *b*)

events Module

Dirt Simple Events

A Dispatcher (or a subclass of Dispatcher) stores event handlers that are ‘fired’ simple event objects when interesting things happen.

Create a dispatcher:

```
>>> d = Dispatcher()
```

Now create a handler for the event and subscribe it to the dispatcher to handle Event events. A handler is a simple function or method that accepts the event as an argument:

```
>>> def handler1(event): print(repr(event))
>>> d.subscribe(Event, handler1)
```

Now dispatch a new event into the dispatcher, and see handler1 get fired:

```
>>> d.dispatch(Event(foo='bar', data='yours', used_by='the event handlers'))
<rdflib.events.Event ['data', 'foo', 'used_by']>
```

class rdflib.events.**Event**(***kw*)
Bases: object

An event is a container for attributes. The source of an event creates this object, or a subclass, gives it any kind of data that the events handlers need to handle the event, and then calls notify(event).

The target of an event registers a function to handle the event it is interested with subscribe(). When a sources calls notify(event), each subscriber to that event will be called in no particular order.

__init__(***kw*)

__module__ = 'rdflib.events'

__repr__()

class rdflib.events.**Dispatcher**
Bases: object

An object that can dispatch events to a privately managed group of subscribers.

__module__ = 'rdflib.events'

dispatch(*event*)
Dispatch the given event to the subscribed handlers for the event's type

get_map()

set_map(*amap*)

subscribe(*event_type*, *handler*)
Subscribe the given handler to an event_type. Handlers are called in the order they are subscribed.

exceptions Module

TODO:

exception rdflib.exceptions.**Error**(*msg=None*)
Bases: exceptions.Exception

Base class for rdflib exceptions.

__init__(*msg=None*)

__module__ = 'rdflib.exceptions'

__weakref__
list of weak references to the object (if defined)

exception rdflib.exceptions.**TypeCheckError**(*node*)
Bases: *rdflib.exceptions.Error*

Parts of assertions are subject to type checks.

__init__(*node*)

__module__ = 'rdflib.exceptions'

exception rdflib.exceptions.**SubjectTypeError**(*node*)
Bases: *rdflib.exceptions.TypeCheckError*

Subject of an assertion must be an instance of URIRef.

__init__(*node*)

__module__ = 'rdflib.exceptions'

exception rdflib.exceptions.**PredicateTypeError**(*node*)
Bases: *rdflib.exceptions.TypeCheckError*

Predicate of an assertion must be an instance of URIRef.

__init__(*node*)

__module__ = 'rdflib.exceptions'

exception rdflib.exceptions.**ObjectTypeError**(*node*)
Bases: *rdflib.exceptions.TypeCheckError*

Object of an assertion must be an instance of URIRef, Literal, or BNode.

__init__(*node*)

__module__ = 'rdflib.exceptions'

exception rdflib.exceptions.**ContextTypeError**(*node*)
Bases: *rdflib.exceptions.TypeCheckError*

Context of an assertion must be an instance of URIRef.

__init__(*node*)

__module__ = 'rdflib.exceptions'

exception rdflib.exceptions.**ParserError**(*msg*)
Bases: *rdflib.exceptions.Error*

RDF Parser error.

__init__(*msg*)

__module__ = 'rdflib.exceptions'

__str__()

graph Module

RDFLib defines the following kinds of Graphs:

- *Graph*
- *QuotedGraph*
- *ConjunctiveGraph*
- *Dataset*

Graph

An RDF graph is a set of RDF triples. Graphs support the python `in` operator, as well as iteration and some operations like union, difference and intersection.

see *Graph*

Conjunctive Graph

A Conjunctive Graph is the most relevant collection of graphs that are considered to be the boundary for closed world assumptions. This boundary is equivalent to that of the store instance (which is itself uniquely identified and distinct from other instances of `Store` that signify other Conjunctive Graphs). It is equivalent to all the named graphs within it and associated with a `_default_` graph which is automatically assigned a `BNode` for an identifier - if one isn't given.

see *ConjunctiveGraph*

Quoted graph

The notion of an RDF graph [14] is extended to include the concept of a formula node. A formula node may occur wherever any other kind of node can appear. Associated with a formula node is an RDF graph that is completely disjoint from all other graphs; i.e. has no nodes in common with any other graph. (It may contain the same labels as other RDF graphs; because this is, by definition, a separate graph, considerations of tidiness do not apply between the graph at a formula node and any other graph.)

This is intended to map the idea of "{ N3-expression }" that is used by N3 into an RDF graph upon which RDF semantics is defined.

see *QuotedGraph*

Dataset

The RDF 1.1 Dataset, a small extension to the Conjunctive Graph. The primary term is "graphs in the datasets" and not "contexts with quads" so there is a separate method to set/retrieve a graph in a dataset and to operate with dataset graphs. As a consequence of this approach, dataset graphs cannot be identified with blank nodes, a name is always required (RDFLib will automatically add a name if one is not provided at creation time). This implementation includes a convenience method to directly add a single quad to a dataset graph.

see *Dataset*

Working with graphs Instantiating Graphs with default store (IOMemory) and default identifier (a BNode):

```
>>> g = Graph()
>>> g.store.__class__
<class 'rdflib.plugins.memory.IOMemory'>
>>> g.identifier.__class__
<class 'rdflib.term.BNode'>
```

Instantiating Graphs with a IOMemory store and an identifier - <http://rdflib.net>:

```
>>> g = Graph('IOMemory', URIRef("http://rdflib.net"))
>>> g.identifier
rdflib.term.URIRef(u'http://rdflib.net')
>>> str(g)
```

```
"<http://rdflib.net> a rdfg:Graph;rdflib:storage
 [a rdflib:Store;rdfs:label 'IOMemory'];"
```

Creating a ConjunctiveGraph - The top level container for all named Graphs in a 'database':

```
>>> g = ConjunctiveGraph()
>>> str(g.default_context)
"[a rdfg:Graph;rdflib:storage [a rdflib:Store;rdfs:label 'IOMemory']];"
```

Adding / removing reified triples to Graph and iterating over it directly or via triple pattern:

```
>>> g = Graph()
>>> statementId = BNode()
>>> print(len(g))
0
>>> g.add((statementId, RDF.type, RDF.Statement))
>>> g.add((statementId, RDF.subject,
...     URIRef(u'http://rdflib.net/store/ConjunctiveGraph')))
>>> g.add((statementId, RDF.predicate, RDFS.label))
>>> g.add((statementId, RDF.object, Literal("Conjunctive Graph")))
>>> print(len(g))
4
>>> for s, p, o in g:
...     print(type(s))
...
<class 'rdflib.term.BNode'>
<class 'rdflib.term.BNode'>
<class 'rdflib.term.BNode'>
<class 'rdflib.term.BNode'>
```

```
>>> for s, p, o in g.triples((None, RDF.object, None)):
...     print(o)
...
Conjunctive Graph
>>> g.remove((statementId, RDF.type, RDF.Statement))
>>> print(len(g))
3
```

`None` terms in calls to *`triples()`* can be thought of as "open variables".

Graph support set-theoretic operators, you can add/subtract graphs, as well as intersection (with multiplication operator g1*g2) and xor (g1 ^ g2).

Note that BNode IDs are kept when doing set-theoretic operations, this may or may not be what you want. Two named graphs within the same application probably want share BNode IDs, two graphs with data from different sources probably not. If your BNode IDs are all generated by RDFLib they are UUIDs and unique.

```
>>> g1 = Graph()
>>> g2 = Graph()
>>> u = URIRef(u'http://example.com/foo')
>>> g1.add([u, RDFS.label, Literal('foo')])
>>> g1.add([u, RDFS.label, Literal('bar')])
>>> g2.add([u, RDFS.label, Literal('foo')])
>>> g2.add([u, RDFS.label, Literal('bing')])
>>> len(g1 + g2)  # adds bing as label
3
>>> len(g1 - g2)  # removes foo
1
>>> len(g1 * g2)  # only foo
```

```
1
>>> g1 += g2  # now g1 contains everything
```

Graph Aggregation - ConjunctiveGraphs and ReadOnlyGraphAggregate within the same store:

```
>>> store = plugin.get('IOMemory', Store)()
>>> g1 = Graph(store)
>>> g2 = Graph(store)
>>> g3 = Graph(store)
>>> stmt1 = BNode()
>>> stmt2 = BNode()
>>> stmt3 = BNode()
>>> g1.add((stmt1, RDF.type, RDF.Statement))
>>> g1.add((stmt1, RDF.subject,
...     URIRef(u'http://rdflib.net/store/ConjunctiveGraph')))
>>> g1.add((stmt1, RDF.predicate, RDFS.label))
>>> g1.add((stmt1, RDF.object, Literal("Conjunctive Graph")))
>>> g2.add((stmt2, RDF.type, RDF.Statement))
>>> g2.add((stmt2, RDF.subject,
...     URIRef(u'http://rdflib.net/store/ConjunctiveGraph')))
>>> g2.add((stmt2, RDF.predicate, RDF.type))
>>> g2.add((stmt2, RDF.object, RDFS.Class))
>>> g3.add((stmt3, RDF.type, RDF.Statement))
>>> g3.add((stmt3, RDF.subject,
...     URIRef(u'http://rdflib.net/store/ConjunctiveGraph')))
>>> g3.add((stmt3, RDF.predicate, RDFS.comment))
>>> g3.add((stmt3, RDF.object, Literal(
...     "The top-level aggregate graph - The sum " +
...     "of all named graphs within a Store")))
>>> len(list(ConjunctiveGraph(store).subjects(RDF.type, RDF.Statement)))
3
>>> len(list(ReadOnlyGraphAggregate([g1,g2]).subjects(
...     RDF.type, RDF.Statement)))
2
```

ConjunctiveGraphs have a *quads()* method which returns quads instead of triples, where the fourth item is the Graph (or subclass thereof) instance in which the triple was asserted:

```
>>> uniqueGraphNames = set(
...     [graph.identifier for s, p, o, graph in ConjunctiveGraph(store
...     ).quads((None, RDF.predicate, None))])
>>> len(uniqueGraphNames)
3
>>> unionGraph = ReadOnlyGraphAggregate([g1, g2])
>>> uniqueGraphNames = set(
...     [graph.identifier for s, p, o, graph in unionGraph.quads(
...     (None, RDF.predicate, None))])
>>> len(uniqueGraphNames)
2
```

Parsing N3 from a string

```
>>> g2 = Graph()
>>> src = '''
... @prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
... @prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
... [ a rdf:Statement ;
...   rdf:subject <http://rdflib.net/store#ConjunctiveGraph>;
...   rdf:predicate rdfs:label;
```

```
...   rdf:object "Conjunctive Graph" ] .
... '''
>>> g2 = g2.parse(data=src, format='n3')
>>> print(len(g2))
4
```

Using Namespace class:

```
>>> RDFLib = Namespace('http://rdflib.net/')
>>> RDFLib.ConjunctiveGraph
rdflib.term.URIRef(u'http://rdflib.net/ConjunctiveGraph')
>>> RDFLib['Graph']
rdflib.term.URIRef(u'http://rdflib.net/Graph')
```

class rdflib.graph.**Graph**(*store='default'*, *identifier=None*, *namespace_manager=None*)
Bases: *rdflib.term.Node*

An RDF Graph

The constructor accepts one argument, the 'store' that will be used to store the graph data (see the 'store' package for stores currently shipped with rdflib).

Stores can be context-aware or unaware. Unaware stores take up (some) less space but cannot support features that require context, such as true merging/demerging of sub-graphs and provenance.

The Graph constructor can take an identifier which identifies the Graph by name. If none is given, the graph is assigned a BNode for its identifier. For more on named graphs, see: http://www.w3.org/2004/03/trix/

__add__(*other*)
Set-theoretic union BNode IDs are not changed.

__and__(*other*)
Set-theoretic intersection. BNode IDs are not changed.

__cmp__(*other*)

__contains__(*triple*)
Support for 'triple in graph' syntax

__eq__(*other*)

__ge__(*other*)

__getitem__(*item*)
A graph can be "sliced" as a shortcut for the triples method The python slice syntax is (ab)used for specifying triples. A generator over matches is returned, the returned tuples include only the parts not given

```
>>> import rdflib
>>> g = rdflib.Graph()
>>> g.add((rdflib.URIRef('urn:bob'), rdflib.RDFS.label, rdflib.Literal('Bob')))
```

```
>>> list(g[rdflib.URIRef('urn:bob')]) # all triples about bob
[(rdflib.term.URIRef(u'http://www.w3.org/2000/01/rdf-schema#label'), rdflib.term.Literal(u'B
```

```
>>> list(g[:rdflib.RDFS.label]) # all label triples
[(rdflib.term.URIRef(u'urn:bob'), rdflib.term.Literal(u'Bob'))]
```

```
>>> list(g[::rdflib.Literal('Bob')]) # all triples with bob as object
[(rdflib.term.URIRef(u'urn:bob'), rdflib.term.URIRef(u'http://www.w3.org/2000/01/rdf-schema#
```

Combined with SPARQL paths, more complex queries can be written concisely:

Name of all Bobs friends:

g[bob : FOAF.knows/FOAF.name]

Some label for Bob:

g[bob : DC.title|FOAF.name|RDFS.label]

All friends and friends of friends of Bob

g[bob : FOAF.knows * '+']

etc.

New in version 4.0.

__gt__(*other*)

__hash__()

__iadd__(*other*)
Add all triples in Graph other to Graph. BNode IDs are not changed.

__init__(*store='default'*, *identifier=None*, *namespace_manager=None*)

__isub__(*other*)
Subtract all triples in Graph other from Graph. BNode IDs are not changed.

__iter__()
Iterates over all triples in the store

__le__(*other*)

__len__()
Returns the number of triples in the graph

If context is specified then the number of triples in the context is returned instead.

__lt__(*other*)

__module__ = 'rdflib.graph'

__mul__(*other*)
Set-theoretic intersection. BNode IDs are not changed.

__or__(*other*)
Set-theoretic union BNode IDs are not changed.

__reduce__()

__repr__()

__str__()

__sub__(*other*)
Set-theoretic difference. BNode IDs are not changed.

__xor__(*other*)
Set-theoretic XOR. BNode IDs are not changed.

absolutize(*uri*, *defrag=1*)
Turn uri into an absolute URI if it's not one already

add(*(s, p, o)*)
Add a triple with self as context

addN(*quads*)
Add a sequence of triple with context

all_nodes()

bind(*prefix*, *namespace*, *override=True*)
Bind prefix to namespace

If override is True will bind namespace to given prefix even if namespace was already bound to a different prefix.

for example: graph.bind('foaf', 'http://xmlns.com/foaf/0.1/')

close(*commit_pending_transaction=False*)
Close the graph store

Might be necessary for stores that require closing a connection to a database or releasing some resource.

comment(*subject*, *default=''*)
Query for the RDFS.comment of the subject

Return default if no comment exists

commit()
Commits active transactions

compute_qname(*uri*, *generate=True*)

connected()
Check if the Graph is connected

The Graph is considered undirectional.

Performs a search on the Graph, starting from a random node. Then iteratively goes depth-first through the triplets where the node is subject and object. Return True if all nodes have been visited and False if it cannot continue and there are still unvisited nodes left.

de_skolemize(*new_graph=None*, *uriref=None*)

destroy(*configuration*)
Destroy the store identified by *configuration* if supported

identifier

isomorphic(*other*)
does a very basic check if these graphs are the same If no BNodes are involved, this is accurate.

See rdflib.compare for a correct implementation of isomorphism checks

items(*list*)
Generator over all items in the resource specified by list

list is an RDF collection.

label(*subject*, *default=''*)
Query for the RDFS.label of the subject

Return default if no label exists or any label if multiple exist.

load(*source*, *publicID=None*, *format='xml'*)

md5_term_hash()

n3()
return an n3 identifier for the Graph

namespace_manager
this graph's namespace-manager

namespaces()
Generator over all the prefix, namespace tuples

objects(*subject=None*, *predicate=None*)
A generator of objects with the given subject and predicate

open(*configuration*, *create=False*)
Open the graph store

Might be necessary for stores that require opening a connection to a database or acquiring some resource.

parse(*source=None*, *publicID=None*, *format=None*, *location=None*, *file=None*, *data=None*, ***args*)
Parse source adding the resulting triples to the Graph.

The source is specified using one of source, location, file or data.

Parameters

- *source*: An InputSource, file-like object, or string. In the case of a string the string is the location of the source.
- *location*: A string indicating the relative or absolute URL of the source. Graph's absolutize method is used if a relative location is specified.
- *file*: A file-like object.
- *data*: A string containing the data to be parsed.
- *format*: Used if format can not be determined from source. Defaults to rdf/xml. Format support can be extended with plugins, but 'xml', 'n3', 'nt', 'trix', 'rdfa' are built in.
- *publicID*: the logical URI to use as the document base. If None specified the document location is used (at least in the case where there is a document location).

Returns

- self, the graph instance.

Examples:

```
>>> my_data = '''
... <rdf:RDF
...   xmlns:rdf='http://www.w3.org/1999/02/22-rdf-syntax-ns#'
...   xmlns:rdfs='http://www.w3.org/2000/01/rdf-schema#'
... >
...   <rdf:Description>
...     <rdfs:label>Example</rdfs:label>
...     <rdfs:comment>This is really just an example.</rdfs:comment>
...   </rdf:Description>
... </rdf:RDF>
... '''
>>> import tempfile
>>> fd, file_name = tempfile.mkstemp()
>>> f = os.fdopen(fd, 'w')
>>> dummy = f.write(my_data)  # Returns num bytes written on py3
>>> f.close()
```

```
>>> g = Graph()
>>> result = g.parse(data=my_data, format="application/rdf+xml")
>>> len(g)
2
```

```
>>> g = Graph()
>>> result = g.parse(location=file_name, format="application/rdf+xml")
>>> len(g)
2
```

```
>>> g = Graph()
>>> result = g.parse(file=open(file_name, "r"),
...     format="application/rdf+xml")
>>> len(g)
2
```

```
>>> os.remove(file_name)
```

predicate_objects(*subject=None*)
A generator of (predicate, object) tuples for the given subject

predicates(*subject=None*, *object=None*)
A generator of predicates with the given subject and object

preferredLabel(*subject*, *lang=None*, *default=None*, *labelProperties=(rdflib.term.URIRef(u'http://www.w3.org/2004/02/skos/core#prefLabel'), rdflib.term.URIRef(u'http://www.w3.org/2000/01/rdf-schema#label'))*)
Find the preferred label for subject.

By default prefers skos:prefLabels over rdfs:labels. In case at least one prefLabel is found returns those, else returns labels. In case a language string (e.g., 'en', 'de' or even '' for no lang-tagged literals) is given, only such labels will be considered.

Return a list of (labelProp, label) pairs, where labelProp is either skos:prefLabel or rdfs:label.

```
>>> from rdflib import ConjunctiveGraph, URIRef, RDFS, Literal
>>> from rdflib.namespace import SKOS
>>> from pprint import pprint
>>> g = ConjunctiveGraph()
>>> u = URIRef(u'http://example.com/foo')
>>> g.add([u, RDFS.label, Literal('foo')])
>>> g.add([u, RDFS.label, Literal('bar')])
>>> pprint(sorted(g.preferredLabel(u)))
[(rdflib.term.URIRef(u'http://www.w3.org/2000/01/rdf-schema#label'),
  rdflib.term.Literal(u'bar')),
 (rdflib.term.URIRef(u'http://www.w3.org/2000/01/rdf-schema#label'),
  rdflib.term.Literal(u'foo'))]
>>> g.add([u, SKOS.prefLabel, Literal('bla')])
>>> pprint(g.preferredLabel(u))
[(rdflib.term.URIRef(u'http://www.w3.org/2004/02/skos/core#prefLabel'),
  rdflib.term.Literal(u'bla'))]
>>> g.add([u, SKOS.prefLabel, Literal('blubb', lang='en')])
>>> sorted(g.preferredLabel(u))
[(rdflib.term.URIRef(u'http://www.w3.org/2004/02/skos/core#prefLabel'),
  rdflib.term.Literal(u'bla')),
  (rdflib.term.URIRef(u'http://www.w3.org/2004/02/skos/core#prefLabel'),
  rdflib.term.Literal(u'blubb', lang='en'))]
>>> g.preferredLabel(u, lang='')
[(rdflib.term.URIRef(u'http://www.w3.org/2004/02/skos/core#prefLabel'),
  rdflib.term.Literal(u'bla'))]
>>> pprint(g.preferredLabel(u, lang='en'))
[(rdflib.term.URIRef(u'http://www.w3.org/2004/02/skos/core#prefLabel'),
  rdflib.term.Literal(u'blubb', lang='en'))]
```

qname(*uri*)

query(*query_object*, *processor='sparql'*, *result='sparql'*, *initNs=None*, *initBindings=None*, *use_store_provided=True*, ***kwargs*)
Query this graph.

A type of 'prepared queries' can be realised by providing initial variable bindings with initBindings

Initial namespaces are used to resolve prefixes used in the query, if none are given, the namespaces from the graph's namespace manager are used.

Returntype rdflib.query.QueryResult

remove(*(s, p, o)*)
Remove a triple from the graph

If the triple does not provide a context attribute, removes the triple from all contexts.

resource(*identifier*)
Create a new `Resource` instance.

Parameters:

- `identifier`: a URIRef or BNode instance.

Example:

```
>>> graph = Graph()
>>> uri = URIRef("http://example.org/resource")
>>> resource = graph.resource(uri)
>>> assert isinstance(resource, Resource)
>>> assert resource.identifier is uri
>>> assert resource.graph is graph
```

rollback()
Rollback active transactions

seq(*subject*)
Check if subject is an rdf:Seq

If yes, it returns a Seq class instance, None otherwise.

serialize(*destination=None*, *format='xml'*, *base=None*, *encoding=None*, ***args*)
Serialize the Graph to destination

If destination is None serialize method returns the serialization as a string. Format defaults to xml (AKA rdf/xml).

Format support can be extended with plugins, but 'xml', 'n3', 'turtle', 'nt', 'pretty-xml', trix' are built in.

set(*triple*)
Convenience method to update the value of object

Remove any existing triples for subject and predicate before adding (subject, predicate, object).

skolemize(*new_graph=None*, *bnode=None*)

store

subject_objects(*predicate=None*)
A generator of (subject, object) tuples for the given predicate

subject_predicates(*object=None*)
A generator of (subject, predicate) tuples for the given object

subjects(*predicate=None*, *object=None*)
A generator of subjects with the given predicate and object

toPython()

transitiveClosure(*func*, *arg*, *seen=None*)
Generates transitive closure of a user-defined function against the graph

```
>>> from rdflib.collection import Collection
>>> g=Graph()
>>> a=BNode('foo')
>>> b=BNode('bar')
>>> c=BNode('baz')
>>> g.add((a,RDF.first,RDF.type))
>>> g.add((a,RDF.rest,b))
>>> g.add((b,RDF.first,RDFS.label))
>>> g.add((b,RDF.rest,c))
>>> g.add((c,RDF.first,RDFS.comment))
>>> g.add((c,RDF.rest,RDF.nil))
>>> def topList(node,g):
...    for s in g.subjects(RDF.rest,node):
...       yield s
>>> def reverseList(node,g):
...    for f in g.objects(node,RDF.first):
...       print(f)
...    for s in g.subjects(RDF.rest,node):
...       yield s
```

```
>>> [rt for rt in g.transitiveClosure(
...     topList,RDF.nil)]
[rdflib.term.BNode('baz'),
 rdflib.term.BNode('bar'),
 rdflib.term.BNode('foo')]
```

```
>>> [rt for rt in g.transitiveClosure(
...     reverseList,RDF.nil)]
http://www.w3.org/2000/01/rdf-schema#comment
http://www.w3.org/2000/01/rdf-schema#label
http://www.w3.org/1999/02/22-rdf-syntax-ns#type
[rdflib.term.BNode('baz'),
 rdflib.term.BNode('bar'),
 rdflib.term.BNode('foo')]
```

transitive_objects(*subject*, *property*, *remember=None*)
Transitively generate objects for the `property` relationship

Generated objects belong to the depth first transitive closure of the `property` relationship starting at `subject`.

transitive_subjects(*predicate*, *object*, *remember=None*)
Transitively generate objects for the `property` relationship

Generated objects belong to the depth first transitive closure of the `property` relationship starting at `subject`.

triples((*s*, *p*, *o*))
Generator over the triple store

Returns triples that match the given triple pattern. If triple pattern does not provide a context, all contexts will be searched.

triples_choices((*subject*, *predicate*, *object_*), *context=None*)

update(*update_object*, *processor='sparql'*, *initNs={}*, *initBindings={}*, *use_store_provided=True*, ***kwargs*)

value(*subject=None*, *predicate=rdflib.term.URIRef(u'http://www.w3.org/1999/02/22-rdf-syntax-ns#value')*, *object=None*, *default=None*, *any=True*)
Get a value for a pair of two criteria

Exactly one of subject, predicate, object must be None. Useful if one knows that there may only be one value.

It is one of those situations that occur a lot, hence this 'macro' like utility

Parameters: subject, predicate, object – exactly one must be None default – value to be returned if no values found any – if True, return any value in the case there is more than one, else, raise UniquenessError

class rdflib.graph.**ConjunctiveGraph**(*store='default'*, *identifier=None*)
Bases: *rdflib.graph.Graph*

A ConjunctiveGraph is an (unamed) aggregation of all the named graphs in a store.

It has a `default` graph, whose name is associated with the graph throughout its life. *__init__()* can take an identifier to use as the name of this default graph or it will assign a BNode.

All methods that add triples work against this default graph.

All queries are carried out against the union of all graphs.

__contains__(*triple_or_quad*)
Support for 'triple/quad in graph' syntax

__init__(*store='default'*, *identifier=None*)

__len__()
Number of triples in the entire conjunctive graph

__module__ = 'rdflib.graph'

__reduce__()

__str__()

add(*triple_or_quad*)
Add a triple or quad to the store.

if a triple is given it is added to the default context

addN(*quads*)
Add a sequence of triples with context

context_id(*uri*, *context_id=None*)
URI#context

contexts(*triple=None*)
Iterate over all contexts in the graph

If triple is specified, iterate over all contexts the triple is in.

get_context(*identifier*, *quoted=False*)
Return a context graph for the given identifier

identifier must be a URIRef or BNode.

parse(*source=None*, *publicID=None*, *format='xml'*, *location=None*, *file=None*, *data=None*, ***args*)
Parse source adding the resulting triples to its own context (sub graph of this graph).

See *rdflib.graph.Graph.parse()* for documentation on arguments.

Returns

The graph into which the source was parsed. In the case of n3 it returns the root context.

quads(*triple_or_quad=None*)
Iterate over all the quads in the entire conjunctive graph

remove(*triple_or_quad*)
Removes a triple or quads

if a triple is given it is removed from all contexts

a quad is removed from the given context only

remove_context(*context*)
Removes the given context from the graph

triples(*triple_or_quad*, *context=None*)
Iterate over all the triples in the entire conjunctive graph

For legacy reasons, this can take the context to query either as a fourth element of the quad, or as the explicit context keyword paramater. The kw param takes precedence.

triples_choices(*(s, p, o)*, *context=None*)
Iterate over all the triples in the entire conjunctive graph

class rdflib.graph.**QuotedGraph**(*store*, *identifier*)
Bases: *rdflib.graph.Graph*

Quoted Graphs are intended to implement Notation 3 formulae. They are associated with a required identifier that the N3 parser *must* provide in order to maintain consistent formulae identification for scenarios such as implication and other such processing.

__init__(*store*, *identifier*)

__module__ = 'rdflib.graph'

__reduce__()

__str__()

add(*(s, p, o)*)
Add a triple with self as context

addN(*quads*)
Add a sequence of triple with context

n3()
Return an n3 identifier for the Graph

class rdflib.graph.**Seq**(*graph*, *subject*)
Bases: object

Wrapper around an RDF Seq resource

It implements a container type in Python with the order of the items returned corresponding to the Seq content. It is based on the natural ordering of the predicate names _1, _2, _3, etc, which is the 'implementation' of a sequence in RDF terms.

__getitem__(*index*)
Item given by index from the Seq

__init__(*graph*, *subject*)
Parameters:

- **graph:** the graph containing the Seq
- **subject:** the subject of a Seq. Note that the init does not check whether this is a Seq, this is done in whoever creates this instance!

__iter__()
Generator over the items in the Seq

__len__()
Length of the Seq

__module__ = 'rdflib.graph'

toPython()

exception rdflib.graph.**ModificationException**
Bases: exceptions.Exception

__init__()

__module__ = 'rdflib.graph'

__str__()

class rdflib.graph.**Dataset**(*store='default'*, *default_union=False*)
Bases: *rdflib.graph.ConjunctiveGraph*

RDF 1.1 Dataset. Small extension to the Conjunctive Graph: - the primary term is graphs in the datasets and not contexts with quads, so there is a separate method to set/retrieve a graph in a dataset and operate with graphs - graphs cannot be identified with blank nodes - added a method to directly add a single quad

Examples of usage:

```
>>> # Create a new Dataset
>>> ds = Dataset()
>>> # simple triples goes to default graph
>>> ds.add((URIRef('http://example.org/a'),
...    URIRef('http://www.example.org/b'),
...    Literal('foo')))
>>>
>>> # Create a graph in the dataset, if the graph name has already been
>>> # used, the corresponding graph will be returned
>>> # (ie, the Dataset keeps track of the constituent graphs)
>>> g = ds.graph(URIRef('http://www.example.com/gr'))
>>>
>>> # add triples to the new graph as usual
>>> g.add(
...     (URIRef('http://example.org/x'),
...     URIRef('http://example.org/y'),
...     Literal('bar')) )
>>> # alternatively: add a quad to the dataset -> goes to the graph
>>> ds.add(
...     (URIRef('http://example.org/x'),
...     URIRef('http://example.org/z'),
...     Literal('foo-bar'),g) )
>>>
>>> # querying triples return them all regardless of the graph
>>> for t in ds.triples((None,None,None)):
...     print(t)
(rdflib.term.URIRef(u'http://example.org/a'),
 rdflib.term.URIRef(u'http://www.example.org/b'),
 rdflib.term.Literal(u'foo'))
(rdflib.term.URIRef(u'http://example.org/x'),
 rdflib.term.URIRef(u'http://example.org/z'),
 rdflib.term.Literal(u'foo-bar'))
(rdflib.term.URIRef(u'http://example.org/x'),
 rdflib.term.URIRef(u'http://example.org/y'),
 rdflib.term.Literal(u'bar'))
>>>
>>> # querying quads return quads; the fourth argument can be unrestricted
```

```
    >>> # or restricted to a graph
    >>> for q in ds.quads((None, None, None, None)):
    ...     print(q)
    (rdflib.term.URIRef(u'http://example.org/a'),
     rdflib.term.URIRef(u'http://www.example.org/b'),
     rdflib.term.Literal(u'foo'),
     None)
    (rdflib.term.URIRef(u'http://example.org/x'),
     rdflib.term.URIRef(u'http://example.org/y'),
     rdflib.term.Literal(u'bar'),
     rdflib.term.URIRef(u'http://www.example.com/gr'))
    (rdflib.term.URIRef(u'http://example.org/x'),
     rdflib.term.URIRef(u'http://example.org/z'),
     rdflib.term.Literal(u'foo-bar'),
     rdflib.term.URIRef(u'http://www.example.com/gr'))
    >>>
    >>> for q in ds.quads((None,None,None,g)):
    ...     print(q)
    (rdflib.term.URIRef(u'http://example.org/x'),
     rdflib.term.URIRef(u'http://example.org/y'),
     rdflib.term.Literal(u'bar'),
     rdflib.term.URIRef(u'http://www.example.com/gr'))
    (rdflib.term.URIRef(u'http://example.org/x'),
     rdflib.term.URIRef(u'http://example.org/z'),
     rdflib.term.Literal(u'foo-bar'),
     rdflib.term.URIRef(u'http://www.example.com/gr'))
    >>> # Note that in the call above -
    >>> # ds.quads((None,None,None,'http://www.example.com/gr'))
    >>> # would have been accepted, too
    >>>
    >>> # graph names in the dataset can be queried:
    >>> for c in ds.graphs():
    ...     print(c)  # doctest:
    DEFAULT
    http://www.example.com/gr
    >>> # A graph can be created without specifying a name; a skolemized genid
    >>> # is created on the fly
    >>> h = ds.graph()
    >>> for c in ds.graphs():
    ...     print(c)
    DEFAULT
    http://rdlib.net/.well-known/genid/rdflib/N...
    http://www.example.com/gr
    >>> # Note that the Dataset.graphs() call returns names of empty graphs,
    >>> # too. This can be restricted:
    >>> for c in ds.graphs(empty=False):
    ...     print(c)
    DEFAULT
    http://www.example.com/gr
    >>>
    >>> # a graph can also be removed from a dataset via ds.remove_graph(g)
```

New in version 4.0.

__init__(*store='default'*, *default_union=False*)

__module__ = 'rdflib.graph'

__str__()

add_graph(*g*)
alias of graph for consistency

contexts(*triple=None*)

graph(*identifier=None*)

graphs(*triple=None*)

parse(*source=None*, *publicID=None*, *format='xml'*, *location=None*, *file=None*, *data=None*, ***args*)

quads(*quad*)

remove_graph(*g*)

exception rdflib.graph.**UnSupportedAggregateOperation**
Bases: exceptions.Exception

__init__()

__module__ = 'rdflib.graph'

__str__()

class rdflib.graph.**ReadOnlyGraphAggregate**(*graphs*, *store='default'*)
Bases: *rdflib.graph.ConjunctiveGraph*

Utility class for treating a set of graphs as a single graph

Only read operations are supported (hence the name). Essentially a ConjunctiveGraph over an explicit subset of the entire store.

__cmp__(*other*)

__contains__(*triple_or_quad*)

__hash__()

__iadd__(*other*)

__init__(*graphs*, *store='default'*)

__isub__(*other*)

__len__()

__module__ = 'rdflib.graph'

__reduce__()

__repr__()

absolutize(*uri*, *defrag=1*)

add(*(s, p, o)*)

addN(*quads*)

bind(*prefix*, *namespace*, *override=True*)

close()

commit()

compute_qname(*uri*, *generate=True*)

destroy(*configuration*)

n3()

namespaces()

open(*configuration*, *create=False*)

parse(*source*, *publicID=None*, *format='xml'*, ***args*)

qname(*uri*)

quads((*s*, *p*, *o*))
Iterate over all the quads in the entire aggregate graph

remove((*s*, *p*, *o*))

rollback()

triples((*s*, *p*, *o*))

triples_choices((*subject*, *predicate*, *object_*), *context=None*)

namespace Module

Namespace Utilities

RDFLib provides mechanisms for managing Namespaces.

In particular, there is a *Namespace* class that takes as its argument the base URI of the namespace.

```
>>> from rdflib.namespace import Namespace
>>> owl = Namespace('http://www.w3.org/2002/07/owl#')
```

Fully qualified URIs in the namespace can be constructed either by attribute or by dictionary access on Namespace instances:

```
>>> owl.seeAlso
rdflib.term.URIRef(u'http://www.w3.org/2002/07/owl#seeAlso')
>>> owl['seeAlso']
rdflib.term.URIRef(u'http://www.w3.org/2002/07/owl#seeAlso')
```

Automatic handling of unknown predicates As a programming convenience, a namespace binding is automatically created when *rdflib.term.URIRef* predicates are added to the graph.

Importable namespaces The following namespaces are available by directly importing from rdflib:

- RDF
- RDFS
- OWL
- XSD
- FOAF
- SKOS
- DOAP
- DC
- DCTERMS
- VOID

```
>>> from rdflib import OWL
>>> OWL.seeAlso
rdflib.term.URIRef(u'http://www.w3.org/2002/07/owl#seeAlso')
```

rdflib.namespace.**is_ncname**(*name*)

rdflib.namespace.**split_uri**(*uri*)

class rdflib.namespace.**Namespace**
Bases: unicode

Utility class for quickly generating URIRefs with a common prefix

```
>>> from rdflib import Namespace
>>> n = Namespace("http://example.org/")
>>> n.Person # as attribute
rdflib.term.URIRef(u'http://example.org/Person')
>>> n['first-name'] # as item - for things that are not valid python identifiers
rdflib.term.URIRef(u'http://example.org/first-name')
```

__getattr__(*name*)

__getitem__(*key*, *default=None*)

__module__ = 'rdflib.namespace'

static __new__(*value*)

__repr__()

term(*name*)

title

class rdflib.namespace.**ClosedNamespace**(*uri*, *terms*)
Bases: object

A namespace with a closed list of members

Trying to create terms not listen is an error

__getattr__(*name*)

__getitem__(*key*, *default=None*)

__init__(*uri*, *terms*)

__module__ = 'rdflib.namespace'

__repr__()

__str__()

term(*name*)

class rdflib.namespace.**NamespaceManager**(*graph*)
Bases: object

Class for managing prefix => namespace mappings

Sample usage from FuXi ...

```
ruleStore = N3RuleStore(additionalBuiltins=additionalBuiltins)
nsMgr = NamespaceManager(Graph(ruleStore))
ruleGraph = Graph(ruleStore,namespace_manager=nsMgr)
```

and ...

```
>>> import rdflib
>>> from rdflib import Graph
>>> from rdflib.namespace import Namespace, NamespaceManager
>>> exNs = Namespace('http://example.com/')
>>> namespace_manager = NamespaceManager(Graph())
>>> namespace_manager.bind('ex', exNs, override=False)
>>> g = Graph()
>>> g.namespace_manager = namespace_manager
>>> all_ns = [n for n in g.namespace_manager.namespaces()]
>>> assert ('ex', rdflib.term.URIRef('http://example.com/')) in all_ns
>>>
```

__init__(*graph*)

__module__ = 'rdflib.namespace'

absolutize(*uri*, *defrag=1*)

bind(*prefix*, *namespace*, *override=True*, *replace=False*)
bind a given namespace to the prefix

if override, rebind, even if the given namespace is already bound to another prefix.

if replace, replace any existing prefix with the new namespace

compute_qname(*uri*, *generate=True*)

namespaces()

normalizeUri(*rdfTerm*)
Takes an RDF Term and 'normalizes' it into a QName (using the registered prefix) or (unlike compute_qname) the Notation 3 form for URIs: <...URI...>

qname(*uri*)

reset()

store

parser Module

Parser plugin interface.

This module defines the parser plugin interface and contains other related parser support code.

The module is mainly useful for those wanting to write a parser that can plugin to rdflib. If you are wanting to invoke a parser you likely want to do so through the Graph class parse method.

class rdflib.parser.**Parser**
Bases: object

__init__()

__module__ = 'rdflib.parser'

parse(*source*, *sink*)

class rdflib.parser.**InputSource**(*system_id=None*)
Bases: xml.sax.xmlreader.InputSource, object

TODO:

__init__(*system_id=None*)

__module__ = 'rdflib.parser'

class rdflib.parser.**StringInputSource**(*value*, *system_id=None*)
Bases: *rdflib.parser.InputSource*

TODO:

__init__(*value*, *system_id=None*)

__module__ = 'rdflib.parser'

class rdflib.parser.**URLInputSource**(*system_id=None*, *format=None*)
Bases: *rdflib.parser.InputSource*

TODO:

__init__(*system_id=None*, *format=None*)

__module__ = 'rdflib.parser'

__repr__()

class rdflib.parser.**FileInputSource**(*file*)
Bases: *rdflib.parser.InputSource*

__init__(*file*)

__module__ = 'rdflib.parser'

__repr__()

paths Module

This module implements the SPARQL 1.1 Property path operators, as defined in:

http://www.w3.org/TR/sparql11-query/#propertypaths

In SPARQL the syntax is as follows:

Syntax	Matches
iri	An IRI. A path of length one.
^elt	Inverse path (object to subject).
elt1 / elt2	A sequence path of elt1 followed by elt2.
elt1 | elt2	A alternative path of elt1 or elt2 (all possibilities are tried).
elt*	A path that connects the subject and object of the path by zero or more matches of elt.
elt+	A path that connects the subject and object of the path by one or more matches of elt.
elt?	A path that connects the subject and object of the path by zero or one matches of elt.
!iri or !(iri$_1$| ... |iri$_n$)	Negated property set. An IRI which is not one of iri$_1$...iri$_n$. !iri is short for !(iri).
!^iri or !(^iri$_1$| ... |^iri$_n$)	Negated property set where the excluded matches are based on reversed path. That is, not one of iri$_1$...iri$_n$ as reverse paths. !^iri is short for !(^iri).
!(iri$_1$| ...|iri$_j$| ^iri$_{j+1}$| ... |^iri$_n$)	A combination of forward and reverse properties in a negated property set.
(elt)	A group path elt, brackets control precedence.

This module is used internally be the SPARQL engine, but they property paths can also be used to query RDFLib Graphs directly.

Where possible the SPARQL syntax is mapped to python operators, and property path objects can be constructed from existing URIRefs.

```
>>> from rdflib import Graph, Namespace
```

```
>>> foaf=Namespace('http://xmlns.com/foaf/0.1/')
```

```
>>> ~foaf.knows
Path(~http://xmlns.com/foaf/0.1/knows)
```

```
>>> foaf.knows/foaf.name
Path(http://xmlns.com/foaf/0.1/knows / http://xmlns.com/foaf/0.1/name)
```

```
>>> foaf.name|foaf.firstName
Path(http://xmlns.com/foaf/0.1/name | http://xmlns.com/foaf/0.1/firstName)
```

Modifiers (?, , +) *are done using* * *(the multiplication operator) and the strings* ‘‘, ‘?’, ‘+’, also defined as constants in this file.

```
>>> foaf.knows*OneOrMore
Path(http://xmlns.com/foaf/0.1/knows+)
```

The path objects can also be used with the normal graph methods.

First some example data:

```
>>> g=Graph()
```

```
>>> g=g.parse(data='''
... @prefix : <ex:> .
...
... :a :p1 :c ; :p2 :f .
... :c :p2 :e ; :p3 :g .
... :g :p3 :h ; :p2 :j .
... :h :p3 :a ; :p2 :g .
...
... :q :px :q .
...
... ''', format='n3')
```

```
>>> e=Namespace('ex:')
```

Graph contains: >>> (e.a, e.p1/e.p2, e.e) in g True

Graph generator functions, triples, subjects, objects, etc. :

```
>>> list(g.objects(e.c, (e.p3*OneOrMore)/e.p2))
[rdflib.term.URIRef(u'ex:j'), rdflib.term.URIRef(u'ex:g'),
    rdflib.term.URIRef(u'ex:f')]
```

A more complete set of tests:

```
>>> list(evalPath(g, (None, e.p1/e.p2, None)))==[(e.a, e.e)]
True
>>> list(evalPath(g, (e.a, e.p1|e.p2, None)))==[(e.a,e.c), (e.a,e.f)]
True
>>> list(evalPath(g, (e.c, ~e.p1, None))) == [ (e.c, e.a) ]
True
>>> list(evalPath(g, (e.a, e.p1*ZeroOrOne, None))) == [(e.a, e.a), (e.a, e.c)]
True
>>> list(evalPath(g, (e.c, e.p3*OneOrMore, None))) == [
...      (e.c, e.g), (e.c, e.h), (e.c, e.a)]
True
```

```
>>> list(evalPath(g, (e.c, e.p3*ZeroOrMore, None))) == [(e.c, e.c),
...      (e.c, e.g), (e.c, e.h), (e.c, e.a)]
True
>>> list(evalPath(g, (e.a, -e.p1, None))) == [(e.a, e.f)]
True
>>> list(evalPath(g, (e.a, -(e.p1|e.p2), None))) == []
True
>>> list(evalPath(g, (e.g, -~e.p2, None))) == [(e.g, e.j)]
True
>>> list(evalPath(g, (e.e, ~(e.p1/e.p2), None))) == [(e.e, e.a)]
True
>>> list(evalPath(g, (e.a, e.p1/e.p3/e.p3, None))) == [(e.a, e.h)]
True
```

```
>>> list(evalPath(g, (e.q, e.px*OneOrMore, None)))
[(rdflib.term.URIRef(u'ex:q'), rdflib.term.URIRef(u'ex:q'))]
```

```
>>> list(evalPath(g, (None, e.p1|e.p2, e.c)))
[(rdflib.term.URIRef(u'ex:a'), rdflib.term.URIRef(u'ex:c'))]
```

```
>>> list(evalPath(g, (None, ~e.p1, e.a))) == [ (e.c, e.a) ]
True
>>> list(evalPath(g, (None, e.p1*ZeroOrOne, e.c)))
[(rdflib.term.URIRef(u'ex:c'), rdflib.term.URIRef(u'ex:c')),
 (rdflib.term.URIRef(u'ex:a'), rdflib.term.URIRef(u'ex:c'))]
```

```
>>> list(evalPath(g, (None, e.p3*OneOrMore, e.a)))
[(rdflib.term.URIRef(u'ex:h'), rdflib.term.URIRef(u'ex:a')),
 (rdflib.term.URIRef(u'ex:g'), rdflib.term.URIRef(u'ex:a')),
 (rdflib.term.URIRef(u'ex:c'), rdflib.term.URIRef(u'ex:a'))]
```

```
>>> list(evalPath(g, (None, e.p3*ZeroOrMore, e.a)))
[(rdflib.term.URIRef(u'ex:a'), rdflib.term.URIRef(u'ex:a')),
 (rdflib.term.URIRef(u'ex:h'), rdflib.term.URIRef(u'ex:a')),
 (rdflib.term.URIRef(u'ex:g'), rdflib.term.URIRef(u'ex:a')),
 (rdflib.term.URIRef(u'ex:c'), rdflib.term.URIRef(u'ex:a'))]
```

```
>>> list(evalPath(g, (None, -e.p1, e.f))) == [(e.a, e.f)]
True
>>> list(evalPath(g, (None, -(e.p1|e.p2), e.c))) == []
True
>>> list(evalPath(g, (None, -~e.p2, e.j))) == [(e.g, e.j)]
True
>>> list(evalPath(g, (None, ~(e.p1/e.p2), e.a))) == [(e.e, e.a)]
True
>>> list(evalPath(g, (None, e.p1/e.p3/e.p3, e.h))) == [(e.a, e.h)]
True
```

```
>>> list(evalPath(g, (e.q, e.px*OneOrMore, None)))
[(rdflib.term.URIRef(u'ex:q'), rdflib.term.URIRef(u'ex:q'))]
```

```
>>> list(evalPath(g, (e.c, (e.p2|e.p3)*ZeroOrMore, e.j)))
[(rdflib.term.URIRef(u'ex:c'), rdflib.term.URIRef(u'ex:j'))]
```

No vars specified:

```
>>> sorted(list(evalPath(g, (None, e.p3*OneOrMore, None))))
[(rdflib.term.URIRef(u'ex:c'), rdflib.term.URIRef(u'ex:a')),
```

```
 (rdflib.term.URIRef(u'ex:c'), rdflib.term.URIRef(u'ex:g')),
 (rdflib.term.URIRef(u'ex:c'), rdflib.term.URIRef(u'ex:h')),
 (rdflib.term.URIRef(u'ex:g'), rdflib.term.URIRef(u'ex:a')),
 (rdflib.term.URIRef(u'ex:g'), rdflib.term.URIRef(u'ex:h')),
 (rdflib.term.URIRef(u'ex:h'), rdflib.term.URIRef(u'ex:a'))]
```

New in version 4.0.

class rdflib.paths.**AlternativePath**(**args*)
Bases: *rdflib.paths.Path*

__init__(**args*)

__module__ = 'rdflib.paths'

__repr__()

eval(*graph*, *subj=None*, *obj=None*)

class rdflib.paths.**InvPath**(*arg*)
Bases: *rdflib.paths.Path*

__init__(*arg*)

__module__ = 'rdflib.paths'

__repr__()

eval(*graph*, *subj=None*, *obj=None*)

class rdflib.paths.**MulPath**(*path*, *mod*)
Bases: *rdflib.paths.Path*

__init__(*path*, *mod*)

__module__ = 'rdflib.paths'

__repr__()

eval(*graph*, *subj=None*, *obj=None*, *first=True*)

class rdflib.paths.**NegatedPath**(*arg*)
Bases: *rdflib.paths.Path*

__init__(*arg*)

__module__ = 'rdflib.paths'

__repr__()

eval(*graph*, *subj=None*, *obj=None*)

class rdflib.paths.**Path**

__div__(*other*)
sequence path

__invert__(*p*)
inverse path

__module__ = 'rdflib.paths'

__mul__(*p*, *mul*)
cardinality path

__neg__(*p*)
negated path

__or__(*other*)
alternative path

__truediv__(*other*)
sequence path

eval(*graph*, *subj=None*, *obj=None*)

class rdflib.paths.**PathList**
Bases: list

__module__ = 'rdflib.paths'

class rdflib.paths.**SequencePath**(**args*)
Bases: *rdflib.paths.Path*

__init__(**args*)

__module__ = 'rdflib.paths'

__repr__()

eval(*graph*, *subj=None*, *obj=None*)

rdflib.paths.**evalPath**(*graph*, *t*)

rdflib.paths.**inv_path**(*p*)
inverse path

rdflib.paths.**mul_path**(*p*, *mul*)
cardinality path

rdflib.paths.**neg_path**(*p*)
negated path

rdflib.paths.**path_alternative**(*self*, *other*)
alternative path

rdflib.paths.**path_sequence**(*self*, *other*)
sequence path

plugin Module

Plugin support for rdf.

There are a number of plugin points for rdf: parser, serializer, store, query processor, and query result. Plugins can be registered either through setuptools entry_points or by calling rdf.plugin.register directly.

If you have a package that uses a setuptools based setup.py you can add the following to your setup:

```
entry_points = {
    'rdf.plugins.parser': [
        'nt =     rdf.plugins.parsers.nt:NTParser',
        ],
    'rdf.plugins.serializer': [
        'nt =     rdf.plugins.serializers.NTSerializer:NTSerializer',
        ],
    }
```

See the setuptools dynamic discovery of services and plugins for more information.

rdflib.plugin.**register**(*name*, *kind*, *module_path*, *class_name*)
Register the plugin for (name, kind). The module_path and class_name should be the path to a plugin class.

rdflib.plugin.**get**(*name*, *kind*)
Return the class for the specified (name, kind). Raises a PluginException if unable to do so.

rdflib.plugin.**plugins**(*name=None*, *kind=None*)
A generator of the plugins.

Pass in name and kind to filter... else leave None to match all.

exception rdflib.plugin.**PluginException**(*msg=None*)
Bases: *rdflib.exceptions.Error*

__module__ = 'rdflib.plugin'

class rdflib.plugin.**Plugin**(*name*, *kind*, *module_path*, *class_name*)
Bases: object

__init__(*name*, *kind*, *module_path*, *class_name*)

__module__ = 'rdflib.plugin'

getClass()

class rdflib.plugin.**PKGPlugin**(*name*, *kind*, *ep*)
Bases: *rdflib.plugin.Plugin*

__init__(*name*, *kind*, *ep*)

__module__ = 'rdflib.plugin'

getClass()

py3compat Module

Utility functions and objects to ease Python 3 compatibility.

rdflib.py3compat.**ascii**(*stream*)

rdflib.py3compat.**b**(*s*)

rdflib.py3compat.**cast_bytes**(*s*, *enc='utf-8'*)

rdflib.py3compat.**decodeStringEscape**(*s*)
s is byte-string - replace escapes in string

rdflib.py3compat.**decodeUnicodeEscape**(*s*)

s is a unicode string replace

and u00AC unicode escapes

rdflib.py3compat.**format_doctest_out**(*func_or_str*)
Python 2 version "%(u)s'abc'" --> "u'abc'" "%(b)s'abc'" --> "'abc'" "55%(L)s" --> "55L"

Accepts a string or a function, so it can be used as a decorator.

rdflib.py3compat.**sign**(*n*)

rdflib.py3compat.**type_cmp**(*a*, *b*)

query Module

class rdflib.query.**ResultRow**
a single result row allows accessing bindings as attributes or with []

```
>>> from rdflib import URIRef, Variable
>>> rr=ResultRow({ Variable('a'): URIRef('urn:cake') }, [Variable('a')])
```

```
>>> rr[0]
rdflib.term.URIRef(u'urn:cake')
>>> rr[1]
Traceback (most recent call last):
    ...
IndexError: tuple index out of range
```

```
>>> rr.a
rdflib.term.URIRef(u'urn:cake')
>>> rr.b
Traceback (most recent call last):
    ...
AttributeError: b
```

```
>>> rr['a']
rdflib.term.URIRef(u'urn:cake')
>>> rr['b']
Traceback (most recent call last):
    ...
KeyError: 'b'
```

```
>>> rr[Variable('a')]
rdflib.term.URIRef(u'urn:cake')
```

New in version 4.0.

class rdflib.query.**Processor**(*graph*)
Bases: object

Query plugin interface.

This module is useful for those wanting to write a query processor that can plugin to rdf. If you are wanting to execute a query you likely want to do so through the Graph class query method.

__init__(*graph*)

__module__ = 'rdflib.query'

query(*strOrQuery*, *initBindings={}*, *initNs={}*, *DEBUG=False*)

class rdflib.query.**Result**(*type_*)
Bases: object

A common class for representing query result.

There is a bit of magic here that makes this appear like different Python objects, depending on the type of result.

If the type is "SELECT", iterating will yield lists of QueryRow objects

If the type is "ASK", iterating will yield a single bool (or bool(result) will return the same bool)

If the type is "CONSTRUCT" or "DESCRIBE" iterating will yield the triples.

len(result) also works.

__eq__(*other*)

__getattr__(*name*)

__init__(*type_*)

__iter__()

__len__()

__module__ = 'rdflib.query'

__nonzero__()

bindings
a list of variable bindings as dicts

static parse(*source*, *format='xml'*, ***kwargs*)

serialize(*destination=None*, *encoding='utf-8'*, *format='xml'*, ***args*)

class rdflib.query.**ResultParser**
Bases: object

__init__()

__module__ = 'rdflib.query'

parse(*source*, ***kwargs*)
return a Result object

class rdflib.query.**ResultSerializer**(*result*)
Bases: object

__init__(*result*)

__module__ = 'rdflib.query'

serialize(*stream*, *encoding='utf-8'*, ***kwargs*)
return a string properly serialized

exception rdflib.query.**ResultException**
Bases: exceptions.Exception

__module__ = 'rdflib.query'

resource Module

The *Resource* class wraps a *Graph* and a resource reference (i.e. a *rdflib.term.URIRef* or *rdflib.term.BNode*) to support a resource-oriented way of working with a graph.

It contains methods directly corresponding to those methods of the Graph interface that relate to reading and writing data. The difference is that a Resource also binds a resource identifier, making it possible to work without tracking both the graph and a current subject. This makes for a "resource oriented" style, as compared to the triple orientation of the Graph API.

Resulting generators are also wrapped so that any resource reference values (rdflib.term.URIRef's and :class:'rdflib.term.BNode's) are in turn wrapped as Resources.
(Note that this behaviour differs from the corresponding methods in :class:'~rdflib.graph.Graph, where no such conversion takes place.)

Basic Usage Scenario

Start by importing things we need and define some namespaces:

```
>>> from rdflib import *
>>> FOAF = Namespace("http://xmlns.com/foaf/0.1/")
>>> CV = Namespace("http://purl.org/captsolo/resume-rdf/0.2/cv#")
```

Load some RDF data:

```
>>> graph = Graph().parse(format='n3', data='''
... @prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
... @prefix xsd: <http://www.w3.org/2001/XMLSchema#>.
... @prefix foaf: <http://xmlns.com/foaf/0.1/> .
... @prefix cv: <http://purl.org/captsolo/resume-rdf/0.2/cv#> .
...
... @base <http://example.org/> .
...
... </person/some1#self> a foaf:Person;
...     rdfs:comment "Just a Python & RDF hacker."@en;
...     foaf:depiction </images/person/some1.jpg>;
...     foaf:homepage <http://example.net/>;
...     foaf:name "Some Body" .
...
... </images/person/some1.jpg> a foaf:Image;
...     rdfs:label "some 1"@en;
...     rdfs:comment "Just an image"@en;
...     foaf:thumbnail </images/person/some1-thumb.jpg> .
...
... </images/person/some1-thumb.jpg> a foaf:Image .
...
... [] a cv:CV;
...     cv:aboutPerson </person/some1#self>;
...     cv:hasWorkHistory [ cv:employedIn </#company>;
...             cv:startDate "2009-09-04"^^xsd:date ] .
... ''')
```

Create a Resource:

```
>>> person = Resource(
...     graph, URIRef("http://example.org/person/some1#self"))
```

Retrieve some basic facts:

```
>>> person.identifier
rdflib.term.URIRef(u'http://example.org/person/some1#self')

>>> person.value(FOAF.name)
rdflib.term.Literal(u'Some Body')

>>> person.value(RDFS.comment)
rdflib.term.Literal(u'Just a Python & RDF hacker.', lang=u'en')
```

Resources can be sliced (like graphs, but the subject is fixed):

```
>>> for name in person[FOAF.name]:
...     print(name)
Some Body
>>> person[FOAF.name : Literal("Some Body")]
True
```

Resources as unicode are represented by their identifiers as unicode:

```
>>> unicode(person)
u'Resource(http://example.org/person/some1#self'
```

Resource references are also Resources, so you can easily get e.g. a qname for the type of a resource, like:

```
>>> person.value(RDF.type).qname()
u'foaf:Person'
```

Or for the predicates of a resource:

```
>>> sorted(
...     p.qname() for p in person.predicates()
... )
[u'foaf:depiction', u'foaf:homepage',
 u'foaf:name', u'rdf:type', u'rdfs:comment']
```

Follow relations and get more data from their Resources as well:

```
>>> for pic in person.objects(FOAF.depiction):
...     print(pic.identifier)
...     print(pic.value(RDF.type).qname())
...     print(pic.label())
...     print(pic.comment())
...     print(pic.value(FOAF.thumbnail).identifier)
http://example.org/images/person/some1.jpg
foaf:Image
some 1
Just an image
http://example.org/images/person/some1-thumb.jpg

>>> for cv in person.subjects(CV.aboutPerson):
...     work = list(cv.objects(CV.hasWorkHistory))[0]
...     print(work.value(CV.employedIn).identifier)
...     print(work.value(CV.startDate))
http://example.org/#company
2009-09-04
```

It's just as easy to work with the predicates of a resource:

```
>>> for s, p in person.subject_predicates():
...     print(s.value(RDF.type).qname())
...     print(p.qname())
...     for s, o in p.subject_objects():
...         print(s.value(RDF.type).qname())
...         print(o.value(RDF.type).qname())
cv:CV
cv:aboutPerson
cv:CV
foaf:Person
```

This is useful for e.g. inspection:

```
>>> thumb_ref = URIRef("http://example.org/images/person/some1-thumb.jpg")
>>> thumb = Resource(graph, thumb_ref)
>>> for p, o in thumb.predicate_objects():
...     print(p.qname())
...     print(o.qname())
rdf:type
foaf:Image
```

Similarly, adding, setting and removing data is easy:

```
>>> thumb.add(RDFS.label, Literal("thumb"))
>>> print(thumb.label())
thumb
>>> thumb.set(RDFS.label, Literal("thumbnail"))
>>> print(thumb.label())
thumbnail
>>> thumb.remove(RDFS.label)
>>> list(thumb.objects(RDFS.label))
[]
```

Schema Example

With this artificial schema data:

```
>>> graph = Graph().parse(format='n3', data='''
... @prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
... @prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
... @prefix owl: <http://www.w3.org/2002/07/owl#> .
... @prefix v: <http://example.org/def/v#> .
...
... v:Artifact a owl:Class .
...
... v:Document a owl:Class;
...     rdfs:subClassOf v:Artifact .
...
... v:Paper a owl:Class;
...     rdfs:subClassOf v:Document .
...
... v:Choice owl:oneOf (v:One v:Other) .
...
... v:Stuff a rdf:Seq; rdf:_1 v:One; rdf:_2 v:Other .
...
... ''')
```

From this class:

```
>>> artifact = Resource(graph, URIRef("http://example.org/def/v#Artifact"))
```

we can get at subclasses:

```
>>> subclasses = list(artifact.transitive_subjects(RDFS.subClassOf))
>>> [c.qname() for c in subclasses]
[u'v:Artifact', u'v:Document', u'v:Paper']
```

and superclasses from the last subclass:

```
>>> [c.qname() for c in subclasses[-1].transitive_objects(RDFS.subClassOf)]
[u'v:Paper', u'v:Document', u'v:Artifact']
```

Get items from the Choice:

```
>>> choice = Resource(graph, URIRef("http://example.org/def/v#Choice"))
>>> [it.qname() for it in choice.value(OWL.oneOf).items()]
[u'v:One', u'v:Other']
```

And the sequence of Stuff:

```
>>> stuff = Resource(graph, URIRef("http://example.org/def/v#Stuff"))
>>> [it.qname() for it in stuff.seq()]
[u'v:One', u'v:Other']
```

On add, other resources are auto-unboxed:

```
    >>> paper = Resource(graph, URIRef("http://example.org/def/v#Paper"))
    >>> paper.add(RDFS.subClassOf, artifact)
    >>> artifact in paper.objects(RDFS.subClassOf) # checks Resource instance
    True
    >>> (paper._identifier, RDFS.subClassOf, artifact._identifier) in graph
    True
```

Technical Details

Comparison is based on graph and identifier:

```
>>> g1 = Graph()
>>> t1 = Resource(g1, URIRef("http://example.org/thing"))
>>> t2 = Resource(g1, URIRef("http://example.org/thing"))
>>> t3 = Resource(g1, URIRef("http://example.org/other"))
>>> t4 = Resource(Graph(), URIRef("http://example.org/other"))

>>> t1 is t2
False

>>> t1 == t2
True
>>> t1 != t2
False

>>> t1 == t3
False
>>> t1 != t3
True

>>> t3 != t4
True

>>> t3 < t1 and t1 > t3
True
>>> t1 >= t1 and t1 >= t3
True
>>> t1 <= t1 and t3 <= t1
True

>>> t1 < t1 or t1 < t3 or t3 > t1 or t3 > t3
False
```

Hash is computed from graph and identifier:

```
>>> g1 = Graph()
>>> t1 = Resource(g1, URIRef("http://example.org/thing"))

>>> hash(t1) == hash(Resource(g1, URIRef("http://example.org/thing")))
True

>>> hash(t1) == hash(Resource(Graph(), t1.identifier))
```

```
False
>>> hash(t1) == hash(Resource(Graph(), URIRef("http://example.org/thing")))
False
```

The Resource class is suitable as a base class for mapper toolkits. For example, consider this utility for accessing RDF properties via qname-like attributes:

```
>>> class Item(Resource):
...
...     def __getattr__(self, p):
...         return list(self.objects(self._to_ref(*p.split('_', 1))))
...
...     def _to_ref(self, pfx, name):
...         return URIRef(self._graph.store.namespace(pfx) + name)
```

It works as follows:

```
>>> graph = Graph().parse(format='n3', data='''
... @prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
... @prefix foaf: <http://xmlns.com/foaf/0.1/> .
...
... @base <http://example.org/> .
... </person/some1#self>
...     foaf:name "Some Body";
...     foaf:depiction </images/person/some1.jpg> .
... </images/person/some1.jpg> rdfs:comment "Just an image"@en .
... ''')

>>> person = Item(graph, URIRef("http://example.org/person/some1#self"))

>>> print(person.foaf_name[0])
Some Body
```

The mechanism for wrapping references as resources cooperates with subclasses. Therefore, accessing referenced resources automatically creates new `Item` objects:

```
>>> isinstance(person.foaf_depiction[0], Item)
True

>>> print(person.foaf_depiction[0].rdfs_comment[0])
Just an image
```

class rdflib.resource.**Resource**(*graph*, *subject*)

Bases: object

__eq__(*other*)

__ge__(*other*)

__getitem__(*item*)

__gt__(*other*)

__hash__()

__init__(*graph*, *subject*)

__iter__()

__le__(*other*)

__lt__(*other*)

__module__ = 'rdflib.resource'

__ne__(*other*)

__repr__()

__setitem__(*item*, *value*)

__str__()

__unicode__()

add(*p*, *o*)

comment()

graph

identifier

items()

label()

objects(*predicate=None*)

predicate_objects()

predicates(*o=None*)

qname()

remove(*p*, *o=None*)

seq()

set(*p*, *o*)

subject_objects()

subject_predicates()

subjects(*predicate=None*)

transitive_objects(*predicate*, *remember=None*)

transitive_subjects(*predicate*, *remember=None*)

value(*p=rdflib.term.URIRef(u'http://www.w3.org/1999/02/22-rdf-syntax-ns#value')*, *o=None*, *default=None*, *any=True*)

serializer Module

Serializer plugin interface.

This module is useful for those wanting to write a serializer that can plugin to rdflib. If you are wanting to invoke a serializer you likely want to do so through the Graph class serialize method.

TODO: info for how to write a serializer that can plugin to rdflib. See also rdflib.plugin

class rdflib.serializer.**Serializer**(*store*)

Bases: object

__init__(*store*)

__module__ = 'rdflib.serializer'

relativize(*uri*)

serialize(*stream*, *base=None*, *encoding=None*, ***args*)
Abstract method

store Module

rdflib.store

Types of store `Context-aware`: An RDF store capable of storing statements within contexts is considered context-aware. Essentially, such a store is able to partition the RDF model it represents into individual, named, and addressable sub-graphs.

Relevant Notation3 reference regarding formulae, quoted statements, and such: http://www.w3.org/DesignIssues/Notation3.html

`Formula-aware`: An RDF store capable of distinguishing between statements that are asserted and statements that are quoted is considered formula-aware.

`Transaction-capable`: capable of providing transactional integrity to the RDF operations performed on it.

`Graph-aware`: capable of keeping track of empty graphs.

---

class rdflib.store.**StoreCreatedEvent**(***kw*)
Bases: *rdflib.events.Event*

This event is fired when the Store is created, it has the following attribute:

- `configuration`: string used to create the store

__module__ = 'rdflib.store'

class rdflib.store.**TripleAddedEvent**(***kw*)
Bases: *rdflib.events.Event*

This event is fired when a triple is added, it has the following attributes:

- the `triple` added to the graph
- the `context` of the triple, if any
- the `graph` to which the triple was added

__module__ = 'rdflib.store'

class rdflib.store.**TripleRemovedEvent**(***kw*)
Bases: *rdflib.events.Event*

This event is fired when a triple is removed, it has the following attributes:

- the `triple` removed from the graph
- the `context` of the triple, if any
- the `graph` from which the triple was removed

__module__ = 'rdflib.store'

class rdflib.store.**NodePickler**
Bases: `object`

__getstate__()

__init__()

__module__ = 'rdflib.store'

__setstate__(*state*)

dumps(*obj*, *protocol=None*, *bin=None*)

loads(*s*)

register(*object*, *id*)

class rdflib.store.**Store**(*configuration=None*, *identifier=None*)
Bases: object

__init__(*configuration=None*, *identifier=None*)
identifier: URIRef of the Store. Defaults to CWD configuration: string containing infomation open can use to connect to datastore.

__len__(*context=None*)
Number of statements in the store. This should only account for non- quoted (asserted) statements if the context is not specified, otherwise it should return the number of statements in the formula or context given.

Parameters **context** – a graph instance to query or None

__module__ = 'rdflib.store'

add((*subject*, *predicate*, *object)*, *context*, *quoted=False*)
Adds the given statement to a specific context or to the model. The quoted argument is interpreted by formula-aware stores to indicate this statement is quoted/hypothetical It should be an error to not specify a context and have the quoted argument be True. It should also be an error for the quoted argument to be True when the store is not formula-aware.

addN(*quads*)
Adds each item in the list of statements to a specific context. The quoted argument is interpreted by formula-aware stores to indicate this statement is quoted/hypothetical. Note that the default implementation is a redirect to add

add_graph(*graph*)
Add a graph to the store, no effect if the graph already exists. :param graph: a Graph instance

bind(*prefix*, *namespace*)

close(*commit_pending_transaction=False*)
This closes the database connection. The commit_pending_transaction parameter specifies whether to commit all pending transactions before closing (if the store is transactional).

commit()

context_aware = False

contexts(*triple=None*)
Generator over all contexts in the graph. If triple is specified, a generator over all contexts the triple is in.

if store is graph_aware, may also return empty contexts

Returns a generator over Nodes

create(*configuration*)

destroy(*configuration*)
This destroys the instance of the store identified by the configuration string.

formula_aware = False

gc()
Allows the store to perform any needed garbage collection

graph_aware = False

namespace(*prefix*)

namespaces()

node_pickler

open(*configuration*, *create=False*)
Opens the store specified by the configuration string. If create is True a store will be created if it does not already exist. If create is False and a store does not already exist an exception is raised. An exception is also raised if a store exists, but there is insufficient permissions to open the store. This should return one of: VALID_STORE, CORRUPTED_STORE, or NO_STORE

prefix(*namespace*)

query(*query*, *initNs*, *initBindings*, *queryGraph*, ***kwargs*)
If stores provide their own SPARQL implementation, override this.

queryGraph is None, a URIRef or '__UNION__' If None the graph is specified in the query-string/object If URIRef it specifies the graph to query, If '__UNION__' the union of all named graphs should be queried (This is used by ConjunctiveGraphs Values other than None obviously only makes sense for context-aware stores.)

remove(*(subject*, *predicate*, *object)*, *context=None*)
Remove the set of triples matching the pattern from the store

remove_graph(*graph*)
Remove a graph from the store, this shoud also remove all triples in the graph

Parameters **graphid** – a Graph instance

rollback()

transaction_aware = False

triples(*triple_pattern*, *context=None*)
A generator over all the triples matching the pattern. Pattern can include any objects for used for comparing against nodes in the store, for example, REGEXTerm, URIRef, Literal, BNode, Variable, Graph, QuotedGraph, Date? DateRange?

Parameters **context** – A conjunctive query can be indicated by either

providing a value of None, or a specific context can be queries by passing a Graph instance (if store is context aware).

triples_choices(*(subject*, *predicate*, *object_)*, *context=None*)
A variant of triples that can take a list of terms instead of a single term in any slot. Stores can implement this to optimize the response time from the default 'fallback' implementation, which will iterate over each term in the list and dispatch to triples

update(*update*, *initNs*, *initBindings*, *queryGraph*, ***kwargs*)
If stores provide their own (SPARQL) Update implementation, override this.

queryGraph is None, a URIRef or '__UNION__' If None the graph is specified in the query-string/object If URIRef it specifies the graph to query, If '__UNION__' the union of all named graphs should be queried (This is used by ConjunctiveGraphs Values other than None obviously only makes sense for context-aware stores.)

`term` Module

This module defines the different types of terms. Terms are the kinds of objects that can appear in a quoted/asserted triple. This includes those that are core to RDF:

- *Blank Nodes*
- *URI References*
- *Literals* (which consist of a literal value,datatype and language tag)

Those that extend the RDF model into N3:

- *Formulae*
- *Universal Quantifications (Variables)*

And those that are primarily for matching against 'Nodes' in the underlying Graph:

- REGEX Expressions
- Date Ranges
- Numerical Ranges

`rdflib.term.`**`bind`**(*datatype*, *pythontype*, *constructor=None*, *lexicalizer=None*)
register a new datatype<->pythontype binding

Parameters

- **`constructor`** – an optional function for converting lexical forms into a Python instances, if not given the pythontype is used directly
- **`lexicalizer`** – an optinoal function for converting python objects to lexical form, if not given object.__str__ is used

class `rdflib.term.`**`Node`**
Bases: `object`

A Node in the Graph.

`__module__` = **'rdflib.term'**

`__slots__` = ()

class `rdflib.term.`**`Identifier`**
Bases: *`rdflib.term.Node`*, `unicode`

See http://www.w3.org/2002/07/rdf-identifer-terminology/ regarding choice of terminology.

`__eq__`(*other*)
Equality for Nodes.

```
>>> BNode("foo")==None
False
>>> BNode("foo")==URIRef("foo")
False
>>> URIRef("foo")==BNode("foo")
False
>>> BNode("foo")!=URIRef("foo")
True
>>> URIRef("foo")!=BNode("foo")
True
>>> Variable('a')!=URIRef('a')
True
```

```
>>> Variable('a')!=Variable('a')
False
```

__ge__(*other*)

__gt__(*other*)
This implements ordering for Nodes,

This tries to implement this: http://www.w3.org/TR/sparql11-query/#modOrderBy

Variables are not included in the SPARQL list, but they are greater than BNodes and smaller than everything else

__hash__()

__le__(*other*)

__lt__(*other*)

__module__ = 'rdflib.term'

__ne__(*other*)

static __new__(*value*)

__slots__ = ()

eq(*other*)
A "semantic"/interpreted equality function, by default, same as __eq__

neq(*other*)
A "semantic"/interpreted not equal function, by default, same as __ne__

class rdflib.term.**URIRef**
Bases: *rdflib.term.Identifier*

RDF URI Reference: http://www.w3.org/TR/rdf-concepts/#section-Graph-URIref

__add__(*other*)

__div__(*other*)
sequence path

__getnewargs__()

__invert__(*p*)
inverse path

__mod__(*other*)

__module__ = 'rdflib.term'

__mul__(*p*, *mul*)
cardinality path

__neg__(*p*)
negated path

static __new__(*value*, *base=None*)

__or__(*other*)
alternative path

__radd__(*other*)

__reduce__()

__repr__()

__slots__ = ()

__str__()

__truediv__(*other*)
sequence path

de_skolemize()
Create a Blank Node from a skolem URI, in accordance with http://www.w3.org/TR/rdf11-concepts/#section-skolemization. This function accepts only rdflib type skolemization, to provide a round-tripping within the system.

New in version 4.0.

defrag()

md5_term_hash()
a string of hex that will be the same for two URIRefs that are the same. It is not a suitable unique id.

Supported for backwards compatibility; new code should probably just use __hash__

n3(*namespace_manager=None*)
This will do a limited check for valid URIs, essentially just making sure that the string includes no illegal characters (`<, >, ", {, }, |, \, `, ^`)

Parameters **namespace_manager** – if not None, will be used to make up a prefixed name

toPython()

class rdflib.term.**BNode**
Bases: *rdflib.term.Identifier*

Blank Node: http://www.w3.org/TR/rdf-concepts/#section-blank-nodes

__getnewargs__()

__module__ = 'rdflib.term'

static __new__(*value=None*, *_sn_gen=<function _generator>*, *_prefix='N'*)
only store implementations should pass in a value

__reduce__()

__repr__()

__slots__ = ()

__str__()

md5_term_hash()
a string of hex that will be the same for two BNodes that are the same. It is not a suitable unique id.

Supported for backwards compatibility; new code should probably just use __hash__

n3(*namespace_manager=None*)

skolemize(*authority='http://rdlib.net/'*)
Create a URIRef "skolem" representation of the BNode, in accordance with http://www.w3.org/TR/rdf11-concepts/#section-skolemization

New in version 4.0.

toPython()

class rdflib.term.**Literal**

Bases: *rdflib.term.Identifier*

RDF Literal: http://www.w3.org/TR/rdf-concepts/#section-Graph-Literal

The lexical value of the literal is the unicode object The interpreted, datatyped value is available from .value

Language tags must be valid according to :rfc:5646

For valid XSD datatypes, the lexical form is optionally normalized at construction time. Default behaviour is set by rdflib.NORMALIZE_LITERALS and can be overridden by the normalize parameter to __new__

Equality and hashing of Literals are done based on the lexical form, i.e.:

```
>>> from rdflib.namespace import XSD
```

```
>>> Literal('01')!=Literal('1') # clear - strings differ
True
```

but with data-type they get normalized:

```
>>> Literal('01', datatype=XSD.integer)!=Literal('1', datatype=XSD.integer)
False
```

unless disabled:

```
>>> Literal('01', datatype=XSD.integer, normalize=False)!=Literal('1', datatype=XSD.integer)
True
```

Value based comparison is possible:

```
>>> Literal('01', datatype=XSD.integer).eq(Literal('1', datatype=XSD.float))
True
```

The eq method also provides limited support for basic python types:

```
>>> Literal(1).eq(1) # fine - int compatible with xsd:integer
True
>>> Literal('a').eq('b') # fine - str compatible with plain-lit
False
>>> Literal('a', datatype=XSD.string).eq('a') # fine - str compatible with xsd:string
True
>>> Literal('a').eq(1) # not fine, int incompatible with plain-lit
NotImplemented
```

Greater-than/less-than ordering comparisons are also done in value space, when compatible datatypes are used. Incompatible datatypes are ordered by DT, or by lang-tag. For other nodes the ordering is None < BNode < URIRef < Literal

Any comparison with non-rdflib Node are “NotImplemented” In PY2.X some stable order will be made up by python

In PY3 this is an error.

```
>>> from rdflib import Literal, XSD
>>> lit2006 = Literal('2006-01-01',datatype=XSD.date)
>>> lit2006.toPython()
datetime.date(2006, 1, 1)
>>> lit2006 < Literal('2007-01-01',datatype=XSD.date)
True
>>> Literal(datetime.utcnow()).datatype
rdflib.term.URIRef(u'http://www.w3.org/2001/XMLSchema#dateTime')
>>> Literal(1) > Literal(2) # by value
```

```
    False
    >>> Literal(1) > Literal(2.0) # by value
    False
    >>> Literal('1') > Literal(1) # by DT
    True
    >>> Literal('1') < Literal('1') # by lexical form
    False
    >>> Literal('a', lang='en') > Literal('a', lang='fr') # by lang-tag
    False
    >>> Literal(1) > URIRef('foo') # by node-type
    True
```

The > < operators will eat this NotImplemented and either make up an ordering (py2.x) or throw a TypeError (py3k):

```
    >>> Literal(1).__gt__(2.0)
    NotImplemented
```

__abs__()

```
        >>> abs(Literal(-1))
        rdflib.term.Literal(u'1', datatype=rdflib.term.URIRef(u'http://www.w3.org/2001/XMLSchema#int
```

```
        >>> from rdflib.namespace import XSD
        >>> abs( Literal("-1", datatype=XSD.integer))
        rdflib.term.Literal(u'1', datatype=rdflib.term.URIRef(u'http://www.w3.org/2001/XMLSchema#int
```

```
        >>> abs(Literal("1"))
        Traceback (most recent call last):
          File "<stdin>", line 1, in <module>
        TypeError: Not a number; rdflib.term.Literal(u'1')
```

__add__(*val*)

```
        >>> Literal(1) + 1
        rdflib.term.Literal(u'2', datatype=rdflib.term.URIRef(u'http://www.w3.org/2001/XMLSchema#int
        >>> Literal("1") + "1"
        rdflib.term.Literal(u'11')
```

__eq__(*other*)

Literals are only equal to other literals.

"Two literals are equal if and only if all of the following hold: * The strings of the two lexical forms compare equal, character by character. * Either both or neither have language tags. * The language tags, if any, compare equal. * Either both or neither have datatype URIs. * The two datatype URIs, if any, compare equal, character by character." – 6.5.1 Literal Equality (RDF: Concepts and Abstract Syntax)

```
        >>> Literal("1", datatype=URIRef("foo")) == Literal("1", datatype=URIRef("foo"))
        True
        >>> Literal("1", datatype=URIRef("foo")) == Literal("1", datatype=URIRef("foo2"))
        False
```

```
        >>> Literal("1", datatype=URIRef("foo")) == Literal("2", datatype=URIRef("foo"))
        False
        >>> Literal("1", datatype=URIRef("foo")) == "asdf"
        False
        >>> from rdflib import XSD
```

```
>>> Literal('2007-01-01', datatype=XSD.date) == Literal('2007-01-01', datatype=XSD.date)
True
>>> Literal('2007-01-01', datatype=XSD.date) == date(2007, 1, 1)
False
>>> Literal("one", lang="en") == Literal("one", lang="en")
True
>>> Literal("hast", lang='en') == Literal("hast", lang='de')
False
>>> Literal("1", datatype=XSD.integer) == Literal(1)
True
>>> Literal("1", datatype=XSD.integer) == Literal("01", datatype=XSD.integer)
True
```

__ge__(*other*)

__getstate__()

__gt__(*other*)

This implements ordering for Literals, the other comparison methods delegate here

This tries to implement this: http://www.w3.org/TR/sparql11-query/#modOrderBy

In short, Literals with compatible data-types are orderd in value space, i.e. >>> from rdflib import XSD

```
>>> Literal(1)>Literal(2) # int/int
False
>>> Literal(2.0)>Literal(1) # double/int
True
>>> from decimal import Decimal
>>> Literal(Decimal("3.3")) > Literal(2.0) # decimal/double
True
>>> Literal(Decimal("3.3")) < Literal(4.0) # decimal/double
True
>>> Literal('b')>Literal('a') # plain lit/plain lit
True
>>> Literal('b')>Literal('a', datatype=XSD.string) # plain lit/xsd:string
True
```

Incompatible datatype mismatches ordered by DT

```
>>> Literal(1)>Literal("2") # int>string
False
```

Langtagged literals by lang tag >>> Literal(“a”, lang=”en”)>Literal(“a”, lang=”fr”) False

__hash__()

```
>>> from rdflib.namespace import XSD
>>> a = {Literal('1', datatype=XSD.integer):'one'}
>>> Literal('1', datatype=XSD.double) in a
False
```

“Called for the key object for dictionary operations, and by the built-in function hash(). Should return a 32-bit integer usable as a hash value for dictionary operations. The only required property is that objects which compare equal have the same hash value; it is advised to somehow mix together (e.g., using exclusive or) the hash values for the components of the object that also play a part in comparison of objects.” – 3.4.1 Basic customization (Python)

“Two literals are equal if and only if all of the following hold: * The strings of the two lexical forms compare equal, character by character. * Either both or neither have language tags. * The language tags,

if any, compare equal. * Either both or neither have datatype URIs. * The two datatype URIs, if any, compare equal, character by character." – 6.5.1 Literal Equality (RDF: Concepts and Abstract Syntax)

__invert__()

```
>>> ~(Literal(-1))
rdflib.term.Literal(u'0', datatype=rdflib.term.URIRef(u'http://www.w3.org/2001/XMLSchema#int
```

```
>>> from rdflib.namespace import XSD
>>> ~( Literal("-1", datatype=XSD.integer))
rdflib.term.Literal(u'0', datatype=rdflib.term.URIRef(u'http://www.w3.org/2001/XMLSchema#int
```

Not working:

```
>>> ~(Literal("1"))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Not a number; rdflib.term.Literal(u'1')
```

__le__(*other*)

```
>>> from rdflib.namespace import XSD
>>> Literal('2007-01-01T10:00:00', datatype=XSD.dateTime
...     ) <= Literal('2007-01-01T10:00:00', datatype=XSD.dateTime)
True
```

__lt__(*other*)

__module__ = 'rdflib.term'

__neg__()

```
>>> (- Literal(1))
rdflib.term.Literal(u'-1', datatype=rdflib.term.URIRef(u'http://www.w3.org/2001/XMLSchema#in
>>> (- Literal(10.5))
rdflib.term.Literal(u'-10.5', datatype=rdflib.term.URIRef(u'http://www.w3.org/2001/XMLSchema
>>> from rdflib.namespace import XSD
>>> (- Literal("1", datatype=XSD.integer))
rdflib.term.Literal(u'-1', datatype=rdflib.term.URIRef(u'http://www.w3.org/2001/XMLSchema#in
```

```
>>> (- Literal("1"))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Not a number; rdflib.term.Literal(u'1')
>>>
```

static __new__(*lexical_or_value*, *lang=None*, *datatype=None*, *normalize=None*)

__nonzero__()
Is the Literal "True" This is used for if statements, bool(literal), etc.

__pos__()

```
>>> (+ Literal(1))
rdflib.term.Literal(u'1', datatype=rdflib.term.URIRef(u'http://www.w3.org/2001/XMLSchema#int
>>> (+ Literal(-1))
rdflib.term.Literal(u'-1', datatype=rdflib.term.URIRef(u'http://www.w3.org/2001/XMLSchema#in
```

```
>>> from rdflib.namespace import XSD
>>> (+ Literal("-1", datatype=XSD.integer))
rdflib.term.Literal(u'-1', datatype=rdflib.term.URIRef(u'http://www.w3.org/2001/XMLSchema#in
```

```
>>> (+ Literal("1"))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Not a number; rdflib.term.Literal(u'1')
```

__reduce__()

__repr__()

__setstate__(*arg*)

__slots__ = ('language', 'datatype', 'value', '_language', '_datatype', '_value')

__str__()

datatype

eq(*other*)

Compare the value of this literal with something else

Either, with the value of another literal comparisons are then done in literal "value space", and according to the rules of XSD subtype-substitution/type-promotion

OR, with a python object:

basestring objects can be compared with plain-literals, or those with datatype xsd:string

bool objects with xsd:boolean

a int, long or float with numeric xsd types

isodate date,time,datetime objects with xsd:date,xsd:time or xsd:datetime

Any other operations returns NotImplemented

language

md5_term_hash()

a string of hex that will be the same for two Literals that are the same. It is not a suitable unique id.

Supported for backwards compatibility; new code should probably just use __hash__

n3(*namespace_manager=None*)

Returns a representation in the N3 format.

Examples:

```
>>> Literal("foo").n3()
u'"foo"'
```

Strings with newlines or triple-quotes:

```
>>> Literal("foo\nbar").n3()
u'"""foo\nbar"""'

>>> Literal("'\'").n3()
u'"\'\'\'"'

>>> Literal('"""').n3()
u'"\\"\\"\\""'
```

Language:

```
>>> Literal("hello", lang="en").n3()
u'"hello"@en'
```

Datatypes:

```
>>> Literal(1).n3()
u'"1"^^<http://www.w3.org/2001/XMLSchema#integer>'

>>> Literal(1.0).n3()
u'"1.0"^^<http://www.w3.org/2001/XMLSchema#double>'

>>> Literal(True).n3()
u'"true"^^<http://www.w3.org/2001/XMLSchema#boolean>'
```

Datatype and language isn't allowed (datatype takes precedence):

```
>>> Literal(1, lang="en").n3()
u'"1"^^<http://www.w3.org/2001/XMLSchema#integer>'
```

Custom datatype:

```
>>> footype = URIRef("http://example.org/ns#foo")
>>> Literal("1", datatype=footype).n3()
u'"1"^^<http://example.org/ns#foo>'
```

Passing a namespace-manager will use it to abbreviate datatype URIs:

```
>>> from rdflib import Graph
>>> Literal(1).n3(Graph().namespace_manager)
u'"1"^^xsd:integer'
```

neq(*other*)

normalize()

Returns a new literal with a normalised lexical representation of this literal >>> from rdflib import XSD >>> Literal("01", datatype=XSD.integer, normalize=False).normalize() rdflib.term.Literal(u'1', datatype=rdflib.term.URIRef(u'http://www.w3.org/2001/XMLSchema#integer'))

Illegal lexical forms for the datatype given are simply passed on >>> Literal("a", datatype=XSD.integer, normalize=False) rdflib.term.Literal(u'a', datatype=rdflib.term.URIRef(u'http://www.w3.org/2001/XMLSchema#integer'))

toPython()

Returns an appropriate python datatype derived from this RDF Literal

value

class rdflib.term.**Variable**

Bases: *rdflib.term.Identifier*

A Variable - this is used for querying, or in Formula aware graphs, where Variables can stored in the graph

__module__ = 'rdflib.term'

static __new__(*value*)

__reduce__()

__repr__()

__slots__ = ()

md5_term_hash()
a string of hex that will be the same for two Variables that are the same. It is not a suitable unique id.

Supported for backwards compatibility; new code should probably just use __hash__

n3(*namespace_manager=None*)

toPython()

class rdflib.term.**Statement**
Bases: *rdflib.term.Node*, tuple

__module__ = 'rdflib.term'

static __new__(*(subject, predicate, object), context*)

__reduce__()

toPython()

util Module

Some utility functions.

Miscellaneous utilities

- list2set
- first
- uniq
- more_than

Term characterisation and generation

- to_term
- from_n3

Date/time utilities

- date_time
- parse_date_time

Statement and component type checkers

- check_context
- check_subject
- check_predicate
- check_object
- check_statement
- check_pattern

rdflib.util.**list2set**(*seq*)
Return a new list without duplicates. Preserves the order, unlike set(seq)

rdflib.util.**first**(*seq*)
return the first element in a python sequence for graphs, use graph.value instead

rdflib.util.**uniq**(*sequence, strip=0*)
removes duplicate strings from the sequence.

rdflib.util.**more_than**(*sequence*, *number*)
Returns 1 if sequence has more items than number and 0 if not.

rdflib.util.**to_term**(*s*, *default=None*)
Creates and returns an Identifier of type corresponding to the pattern of the given positional argument string `s`:

'' returns the `default` keyword argument value or `None`

'<s>' returns `URIRef(s)` (i.e. without angle brackets)

'"s"' returns `Literal(s)` (i.e. without doublequotes)

'_s' returns `BNode(s)` (i.e. without leading underscore)

rdflib.util.**from_n3**(*s*, *default=None*, *backend=None*, *nsm=None*)
Creates the Identifier corresponding to the given n3 string.

```
>>> from_n3('<http://ex.com/foo>') == URIRef('http://ex.com/foo')
True
>>> from_n3('"foo"@de') == Literal('foo', lang='de')
True
>>> from_n3('"""multi\nline\nstring"""@en') == Literal(
...     'multi\nline\nstring', lang='en')
True
>>> from_n3('42') == Literal(42)
True
>>> from_n3(Literal(42).n3()) == Literal(42)
True
>>> from_n3('"42"^^xsd:integer') == Literal(42)
True
>>> from rdflib import RDFS
>>> from_n3('rdfs:label') == RDFS['label']
True
>>> nsm = NamespaceManager(Graph())
>>> nsm.bind('dbpedia', 'http://dbpedia.org/resource/')
>>> berlin = URIRef('http://dbpedia.org/resource/Berlin')
>>> from_n3('dbpedia:Berlin', nsm=nsm) == berlin
True
```

rdflib.util.**date_time**(*t=None*, *local_time_zone=False*)
http://www.w3.org/TR/NOTE-datetime ex: 1997-07-16T19:20:30Z

```
>>> date_time(1126482850)
'2005-09-11T23:54:10Z'
```

@@ this will change depending on where it is run #>>> date_time(1126482850, local_time_zone=True) #'2005-09-11T19:54:10-04:00'

```
>>> date_time(1)
'1970-01-01T00:00:01Z'
```

```
>>> date_time(0)
'1970-01-01T00:00:00Z'
```

rdflib.util.**parse_date_time**(*val*)
always returns seconds in UTC

tests are written like this to make any errors easier to understand >>> parse_date_time('2005-09-11T23:54:10Z') - 1126482850.0 0.0

```
>>> parse_date_time('2005-09-11T16:54:10-07:00') - 1126482850.0
0.0
```

```
    >>> parse_date_time('1970-01-01T00:00:01Z') - 1.0
    0.0
```

```
    >>> parse_date_time('1970-01-01T00:00:00Z') - 0.0
    0.0
    >>> parse_date_time("2005-09-05T10:42:00") - 1125916920.0
    0.0
```

rdflib.util.**check_context**(*c*)

rdflib.util.**check_subject**(*s*)
Test that s is a valid subject identifier.

rdflib.util.**check_predicate**(*p*)
Test that p is a valid predicate identifier.

rdflib.util.**check_object**(*o*)
Test that o is a valid object identifier.

rdflib.util.**check_statement**(*triple*)

rdflib.util.**check_pattern**(*triple*)

rdflib.util.**guess_format**(*fpath*, *fmap=None*)
Guess RDF serialization based on file suffix. Uses `SUFFIX_FORMAT_MAP` unless `fmap` is provided. Examples:

```
    >>> guess_format('path/to/file.rdf')
    'xml'
    >>> guess_format('path/to/file.owl')
    'xml'
    >>> guess_format('path/to/file.ttl')
    'turtle'
    >>> guess_format('path/to/file.xhtml')
    'rdfa'
    >>> guess_format('path/to/file.svg')
    'rdfa'
    >>> guess_format('path/to/file.xhtml', {'xhtml': 'grddl'})
    'grddl'
```

This also works with just the suffixes, with or without leading dot, and regardless of letter case:

```
    >>> guess_format('.rdf')
    'xml'
    >>> guess_format('rdf')
    'xml'
    >>> guess_format('RDF')
    'xml'
```

rdflib.util.**find_roots**(*graph*, *prop*, *roots=None*)
Find the roots in some sort of transitive hierarchy.

find_roots(graph, rdflib.RDFS.subClassOf) will return a set of all roots of the sub-class hierarchy

Assumes triple of the form (child, prop, parent), i.e. the direction of RDFS.subClassOf or SKOS.broader

rdflib.util.**get_tree**(*graph*, *root*, *prop*, *mapper=<function <lambda>>*, *sortkey=None*, *done=None*, *dir='down'*)
Return a nested list/tuple structure representing the tree built by the transitive property given, starting from the root given

i.e.

get_tree(graph, rdflib.URIRef("http://xmlns.com/foaf/0.1/Person"), rdflib.RDFS.subClassOf)

will return the structure for the subClassTree below person.

dir='down' assumes triple of the form (child, prop, parent), i.e. the direction of RDFS.subClassOf or SKOS.broader Any other dir traverses in the other direction

void Module

rdflib.void.**generateVoID**(*g*, *dataset=None*, *res=None*, *distinctForPartitions=True*)

Returns a new graph with a VoID description of the passed dataset

For more info on Vocabulary of Interlinked Datasets (VoID), see: http://vocab.deri.ie/void

This only makes two passes through the triples (once to detect the types of things)

The tradeoff is that lots of temporary structures are built up in memory meaning lots of memory may be consumed :) I imagine at least a few copies of your original graph.

the distinctForPartitions parameter controls whether distinctSubjects/objects are tracked for each class/propertyPartition this requires more memory again

Subpackages

extras Package

extras Package

cmdlineutils Module

rdflib.extras.cmdlineutils.**main**(*target*, *_help=<function _help>*, *options=''*, *stdin=True*)

A main function for tools that read RDF from files given on commandline or from STDIN (if stdin parameter is true)

describer Module A Describer is a stateful utility for creating RDF statements in a semi-declarative manner. It has methods for creating literal values, rel and rev resource relations (somewhat resembling RDFa).

The *rel* and `rev` methods return a context manager which sets the current about to the referenced resource for the context scope (for use with the `with` statement).

Full example in the `to_rdf` method below:

```
>>> import datetime
>>> from rdflib.graph import Graph
>>> from rdflib.namespace import Namespace, RDFS, FOAF
>>>
>>> ORG_URI = "http://example.org/"
>>>
>>> CV = Namespace("http://purl.org/captsolo/resume-rdf/0.2/cv#")
>>>
>>> class Person(object):
...     def __init__(self):
...         self.first_name = u"Some"
...         self.last_name = u"Body"
...         self.username = "some1"
...         self.presentation = u"Just a Python & RDF hacker."
...         self.image = "/images/persons/" + self.username + ".jpg"
```

```
...         self.site = "http://example.net/"
...         self.start_date = datetime.date(2009, 9, 4)
...     def get_full_name(self):
...         return u" ".join([self.first_name, self.last_name])
...     def get_absolute_url(self):
...         return "/persons/" + self.username
...     def get_thumbnail_url(self):
...         return self.image.replace('.jpg', '-thumb.jpg')
...
...     def to_rdf(self):
...         graph = Graph()
...         graph.bind('foaf', FOAF)
...         graph.bind('cv', CV)
...         lang = 'en'
...         d = Describer(graph, base=ORG_URI)
...         d.about(self.get_absolute_url()+'#person')
...         d.rdftype(FOAF.Person)
...         d.value(FOAF.name, self.get_full_name())
...         d.value(FOAF.firstName, self.first_name)
...         d.value(FOAF.surname, self.last_name)
...         d.rel(FOAF.homepage, self.site)
...         d.value(RDFS.comment, self.presentation, lang=lang)
...         with d.rel(FOAF.depiction, self.image):
...             d.rdftype(FOAF.Image)
...             d.rel(FOAF.thumbnail, self.get_thumbnail_url())
...         with d.rev(CV.aboutPerson):
...             d.rdftype(CV.CV)
...             with d.rel(CV.hasWorkHistory):
...                 d.value(CV.startDate, self.start_date)
...                 d.rel(CV.employedIn, ORG_URI+"#company")
...         return graph
...
>>> person_graph = Person().to_rdf()
>>> expected = Graph().parse(data='''<?xml version="1.0" encoding="utf-8"?>
... <rdf:RDF
...   xmlns:foaf="http://xmlns.com/foaf/0.1/"
...   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
...   xmlns:cv="http://purl.org/captsolo/resume-rdf/0.2/cv#"
...   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">
...   <foaf:Person rdf:about="http://example.org/persons/some1#person">
...     <foaf:name>Some Body</foaf:name>
...     <foaf:firstName>Some</foaf:firstName>
...     <foaf:surname>Body</foaf:surname>
...     <foaf:depiction>
...       <foaf:Image
...         rdf:about=
...             "http://example.org/images/persons/some1.jpg">
...         <foaf:thumbnail
...         rdf:resource=
...             "http://example.org/images/persons/some1-thumb.jpg"/>
...       </foaf:Image>
...     </foaf:depiction>
...     <rdfs:comment xml:lang="en">
...             Just a Python &amp;amp; RDF hacker.
...     </rdfs:comment>
...     <foaf:homepage rdf:resource="http://example.net/"/>
...   </foaf:Person>
...   <cv:CV>
```

```
...      <cv:aboutPerson
...          rdf:resource="http://example.org/persons/some1#person">
...      </cv:aboutPerson>
...      <cv:hasWorkHistory>
...        <rdf:Description>
...          <cv:startDate
...              rdf:datatype="http://www.w3.org/2001/XMLSchema#date"
...              >2009-09-04</cv:startDate>
...          <cv:employedIn rdf:resource="http://example.org/#company"/>
...        </rdf:Description>
...      </cv:hasWorkHistory>
...    </cv:CV>
... </rdf:RDF>
... ''')
>>>
>>> from rdflib.compare import isomorphic
>>> isomorphic(person_graph, expected)
True
```

class rdflib.extras.describer.**Describer**(*graph=None*, *about=None*, *base=None*)

Bases: object

__init__(*graph=None*, *about=None*, *base=None*)

__module__ = 'rdflib.extras.describer'

about(*subject*, ***kws*)

Sets the current subject. Will convert the given object into an URIRef if it's not an Identifier.

Usage:

```
>>> d = Describer()
>>> d._current()
rdflib.term.BNode(...)
>>> d.about("http://example.org/")
>>> d._current()
rdflib.term.URIRef(u'http://example.org/')
```

rdftype(*t*)

Shorthand for setting rdf:type of the current subject.

Usage:

```
>>> from rdflib import URIRef
>>> from rdflib.namespace import RDF, RDFS
>>> d = Describer(about="http://example.org/")
>>> d.rdftype(RDFS.Resource)
>>> (URIRef('http://example.org/'),
...      RDF.type, RDFS.Resource) in d.graph
True
```

rel(*p*, *o=None*, ***kws*)

Set an object for the given property. Will convert the given object into an URIRef if it's not an Identifier. If none is given, a new BNode is used.

Returns a context manager for use in a with block, within which the given object is used as current subject.

Usage:

```
>>> from rdflib import URIRef
>>> from rdflib.namespace import RDF, RDFS
>>> d = Describer(about="/", base="http://example.org/")
>>> _ctxt = d.rel(RDFS.seeAlso, "/about")
>>> d.graph.value(URIRef('http://example.org/'), RDFS.seeAlso)
rdflib.term.URIRef(u'http://example.org/about')

>>> with d.rel(RDFS.seeAlso, "/more"):
...     d.value(RDFS.label, "More")
>>> (URIRef('http://example.org/'), RDFS.seeAlso,
...         URIRef('http://example.org/more')) in d.graph
True
>>> d.graph.value(URIRef('http://example.org/more'), RDFS.label)
rdflib.term.Literal(u'More')
```

rev(*p*, *s=None*, ***kws*)

Same as `rel`, but uses current subject as *object* of the relation. The given resource is still used as subject in the returned context manager.

Usage:

```
>>> from rdflib import URIRef
>>> from rdflib.namespace import RDF, RDFS
>>> d = Describer(about="http://example.org/")
>>> with d.rev(RDFS.seeAlso, "http://example.net/"):
...     d.value(RDFS.label, "Net")
>>> (URIRef('http://example.net/'), RDFS.seeAlso,
...         URIRef('http://example.org/')) in d.graph
True
>>> d.graph.value(URIRef('http://example.net/'), RDFS.label)
rdflib.term.Literal(u'Net')
```

value(*p*, *v*, ***kws*)

Set a literal value for the given property. Will cast the value to an `Literal` if a plain literal is given.

Usage:

```
>>> from rdflib import URIRef
>>> from rdflib.namespace import RDF, RDFS
>>> d = Describer(about="http://example.org/")
>>> d.value(RDFS.label, "Example")
>>> d.graph.value(URIRef('http://example.org/'), RDFS.label)
rdflib.term.Literal(u'Example')
```

rdflib.extras.describer.**cast_identifier**(*ref*, ***kws*)

rdflib.extras.describer.**cast_value**(*v*, ***kws*)

infixowl Module RDFLib Python binding for OWL Abstract Syntax

see: **http://www.w3.org/TR/owl-semantics/syntax.html** http://owl-workshop.man.ac.uk/acceptedLong/submission_9.pdf

3.2.3 Axioms for complete classes without using owl:equivalentClass

Named class description of type 2 (with owl:oneOf) or type 4-6 (with owl:intersectionOf, owl:unionOf or owl:complementOf

Uses Manchester Syntax for __repr__

```
>>> exNs = Namespace('http://example.com/')
>>> namespace_manager = NamespaceManager(Graph())
>>> namespace_manager.bind('ex', exNs, override=False)
>>> namespace_manager.bind('owl', OWL_NS, override=False)
>>> g = Graph()
>>> g.namespace_manager = namespace_manager
```

Now we have an empty graph, we can construct OWL classes in it using the Python classes defined in this module

```
>>> a = Class(exNs.Opera, graph=g)
```

Now we can assert rdfs:subClassOf and owl:equivalentClass relationships (in the underlying graph) with other classes using the 'subClassOf' and 'equivalentClass' descriptors which can be set to a list of objects for the corresponding predicates.

```
>>> a.subClassOf = [exNs.MusicalWork]
```

We can then access the rdfs:subClassOf relationships

```
>>> print(list(a.subClassOf))
[Class: ex:MusicalWork ]
```

This can also be used against already populated graphs:

```
>>> owlGraph = Graph().parse(OWL_NS)
>>> namespace_manager.bind('owl', OWL_NS, override=False)
>>> owlGraph.namespace_manager = namespace_manager
>>> list(Class(OWL_NS.Class, graph=owlGraph).subClassOf)
[Class: rdfs:Class ]
```

Operators are also available. For instance we can add ex:Opera to the extension of the ex:CreativeWork class via the '+=' operator

```
>>> a
Class: ex:Opera SubClassOf: ex:MusicalWork
>>> b = Class(exNs.CreativeWork, graph=g)
>>> b += a
>>> print(sorted(a.subClassOf, key=lambda c:c.identifier))
[Class: ex:CreativeWork , Class: ex:MusicalWork ]
```

And we can then remove it from the extension as well

```
>>> b -= a
>>> a
Class: ex:Opera SubClassOf: ex:MusicalWork
```

Boolean class constructions can also be created with Python operators. For example, The | operator can be used to construct a class consisting of a owl:unionOf the operands:

```
>>> c =  a | b | Class(exNs.Work, graph=g)
>>> c
( ex:Opera OR ex:CreativeWork OR ex:Work )
```

Boolean class expressions can also be operated as lists (using python list operators)

```
>>> del c[c.index(Class(exNs.Work, graph=g))]
>>> c
( ex:Opera OR ex:CreativeWork )
```

The '&' operator can be used to construct class intersection:

```
>>> woman = Class(exNs.Female, graph=g) & Class(exNs.Human, graph=g)
>>> woman.identifier = exNs.Woman
>>> woman
( ex:Female AND ex:Human )
>>> len(woman)
2
```

Enumerated classes can also be manipulated

```
>>> contList = [Class(exNs.Africa, graph=g), Class(exNs.NorthAmerica, graph=g)]
>>> EnumeratedClass(members=contList, graph=g)
{ ex:Africa ex:NorthAmerica }
```

owl:Restrictions can also be instantiated:

```
>>> Restriction(exNs.hasParent, graph=g, allValuesFrom=exNs.Human)
( ex:hasParent ONLY ex:Human )
```

Restrictions can also be created using Manchester OWL syntax in 'colloquial' Python >>> exNs.hasParent | some | Class(exNs.Physician, graph=g) #doctest: +SKIP (ex:hasParent SOME ex:Physician)

```
>>> Property(exNs.hasParent,graph=g) | max | Literal(1)
( ex:hasParent MAX 1 )
```

```
>>> print(g.serialize(format='pretty-xml'))
```

rdflib.extras.infixowl.**AllClasses**(*graph*)

rdflib.extras.infixowl.**AllDifferent**(*members*)
DisjointClasses(' description description { description } ')'

rdflib.extras.infixowl.**AllProperties**(*graph*)

class rdflib.extras.infixowl.**AnnotatableTerms**(*identifier*, *graph=None*, *nameAnnotation=None*, *nameIsLabel=False*)

Bases: *rdflib.extras.infixowl.Individual*

Terms in an OWL ontology with rdfs:label and rdfs:comment

__init__(*identifier*, *graph=None*, *nameAnnotation=None*, *nameIsLabel=False*)

__module__ = 'rdflib.extras.infixowl'

comment

handleAnnotation(*val*)

label

seeAlso

setupACEAnnotations()

class rdflib.extras.infixowl.**BooleanClass**(*identifier=None*, *operator=rdflib.term.URIRef(u'http://www.w3.org/2002/07/owl#intersectionOf')*, *members=None*, *graph=None*)

Bases: *rdflib.extras.infixowl.OWLRDFListProxy*, *rdflib.extras.infixowl.Class*

See: http://www.w3.org/TR/owl-ref/#Boolean

owl:complementOf is an attribute of Class, however

__init__(*identifier=None*, *operator=rdflib.term.URIRef(u'http://www.w3.org/2002/07/owl#intersectionOf')*, *members=None*, *graph=None*)

__module__ = 'rdflib.extras.infixowl'

__or__(*other*)
Adds other to the list and returns self

__repr__()
Returns the Manchester Syntax equivalent for this class

changeOperator(*newOperator*)

copy()
Create a copy of this class

getIntersections = <rdflib.extras.infixowl.Callable instance>

getUnions = <rdflib.extras.infixowl.Callable instance>

isPrimitive()

serialize(*graph*)

class rdflib.extras.infixowl.**Callable**(*anycallable*)

__init__(*anycallable*)

__module__ = 'rdflib.extras.infixowl'

rdflib.extras.infixowl.**CastClass**(*c*, *graph=None*)

class rdflib.extras.infixowl.**Class**(*identifier=None*, *subClassOf=None*, *equivalentClass=None*, *disjointWith=None*, *complementOf=None*, *graph=None*, *skipOWLClassMembership=False*, *comment=None*, *nounAnnotations=None*, *nameAnnotation=None*, *nameIsLabel=False*)

Bases: *rdflib.extras.infixowl.AnnotatableTerms*

'General form' for classes:

The Manchester Syntax (supported in Protege) is used as the basis for the form of this class

See: http://owl-workshop.man.ac.uk/acceptedLong/submission_9.pdf:

[Annotation] 'Class:' classID {Annotation

(('SubClassOf:' ClassExpression) | ('EquivalentTo' ClassExpression) | ('DisjointWith' ClassExpression)) }

Appropriate excerpts from OWL Reference:

".. Subclass axioms provide us with partial definitions: they represent necessary but not sufficient conditions for establishing class membership of an individual."

".. A class axiom may contain (multiple) owl:equivalentClass statements"

"..A class axiom may also contain (multiple) owl:disjointWith statements.."

"..An owl:complementOf property links a class to precisely one class description."

__and__(*other*)
Construct an anonymous class description consisting of the intersection of this class and 'other' and return it

```
>>> exNs = Namespace('http://example.com/')
>>> namespace_manager = NamespaceManager(Graph())
>>> namespace_manager.bind('ex', exNs, override=False)
>>> namespace_manager.bind('owl', OWL_NS, override=False)
```

```
>>> g = Graph()
>>> g.namespace_manager = namespace_manager
```

Chaining 3 intersections

```
>>> female = Class(exNs.Female, graph=g)
>>> human = Class(exNs.Human, graph=g)
>>> youngPerson = Class(exNs.YoungPerson, graph=g)
>>> youngWoman = female & human & youngPerson
>>> youngWoman
ex:YoungPerson THAT ( ex:Female AND ex:Human )
>>> isinstance(youngWoman, BooleanClass)
True
>>> isinstance(youngWoman.identifier, BNode)
True
```

__eq__(*other*)

__hash__()

```
>>> b=Class(OWL_NS.Restriction)
>>> c=Class(OWL_NS.Restriction)
>>> len(set([b,c]))
1
```

__iadd__(*other*)

__init__(*identifier=None*, *subClassOf=None*, *equivalentClass=None*, *disjointWith=None*, *complementOf=None*, *graph=None*, *skipOWLClassMembership=False*, *comment=None*, *nounAnnotations=None*, *nameAnnotation=None*, *nameIsLabel=False*)

__invert__()
Shorthand for Manchester syntax's not operator

__isub__(*other*)

__module__ = 'rdflib.extras.infixowl'

__or__(*other*)
Construct an anonymous class description consisting of the union of this class and 'other' and return it

__repr__(*full=False*, *normalization=True*)
Returns the Manchester Syntax equivalent for this class

annotation

complementOf

disjointWith

equivalentClass

extent

extentQuery

isPrimitive()

parents
computed attributes that returns a generator over taxonomic 'parents' by disjunction, conjunction, and subsumption

```
        >>> from rdflib.util import first
        >>> exNs = Namespace('http://example.com/')
        >>> namespace_manager = NamespaceManager(Graph())
        >>> namespace_manager.bind('ex', exNs, override=False)
        >>> namespace_manager.bind('owl', OWL_NS, override=False)
        >>> g = Graph()
        >>> g.namespace_manager = namespace_manager
        >>> Individual.factoryGraph = g
        >>> brother = Class(exNs.Brother)
        >>> sister = Class(exNs.Sister)
        >>> sibling = brother | sister
        >>> sibling.identifier = exNs.Sibling
        >>> sibling
        ( ex:Brother OR ex:Sister )
        >>> first(brother.parents)
        Class: ex:Sibling EquivalentTo: ( ex:Brother OR ex:Sister )
        >>> parent = Class(exNs.Parent)
        >>> male = Class(exNs.Male)
        >>> father = parent & male
        >>> father.identifier = exNs.Father
        >>> list(father.parents)
        [Class: ex:Parent , Class: ex:Male ]
```

serialize(*graph*)

setupNounAnnotations(*nounAnnotations*)

subClassOf

subSumpteeIds()

class rdflib.extras.infixowl.**ClassNamespaceFactory**

Bases: *rdflib.namespace.Namespace*

__getattr__(*name*)

__getitem__(*key*, *default=None*)

__module__ = 'rdflib.extras.infixowl'

term(*name*)

rdflib.extras.infixowl.**classOrIdentifier**(*thing*)

rdflib.extras.infixowl.**classOrTerm**(*thing*)

rdflib.extras.infixowl.**CommonNSBindings**(*graph*, *additionalNS={}*)

Takes a graph and binds the common namespaces (rdf,rdfs, & owl)

rdflib.extras.infixowl.**ComponentTerms**(*cls*)

Takes a Class instance and returns a generator over the classes that are involved in its definition, ignoring unamed classes

rdflib.extras.infixowl.**DeepClassClear**(*classToPrune*)

Recursively clear the given class, continuing where any related class is an anonymous class

```
    >>> EX = Namespace('http://example.com/')
    >>> namespace_manager = NamespaceManager(Graph())
    >>> namespace_manager.bind('ex', EX, override=False)
    >>> namespace_manager.bind('owl', OWL_NS, override=False)
    >>> g = Graph()
    >>> g.namespace_manager = namespace_manager
    >>> Individual.factoryGraph = g
```

```
>>> classB = Class(EX.B)
>>> classC = Class(EX.C)
>>> classD = Class(EX.D)
>>> classE = Class(EX.E)
>>> classF = Class(EX.F)
>>> anonClass = EX.someProp | some | classD
>>> classF += anonClass
>>> list(anonClass.subClassOf)
[Class: ex:F ]
>>> classA = classE | classF | anonClass
>>> classB += classA
>>> classA.equivalentClass = [Class()]
>>> classB.subClassOf = [EX.someProp | some | classC]
>>> classA
( ex:E OR ex:F OR ( ex:someProp SOME ex:D ) )
>>> DeepClassClear(classA)
>>> classA
(  )
>>> list(anonClass.subClassOf)
[]
>>> classB
Class: ex:B SubClassOf: ( ex:someProp SOME ex:C )
```

```
>>> otherClass = classD | anonClass
>>> otherClass
( ex:D OR ( ex:someProp SOME ex:D ) )
>>> DeepClassClear(otherClass)
>>> otherClass
(  )
>>> otherClass.delete()
>>> list(g.triples((otherClass.identifier, None, None)))
[]
```

class rdflib.extras.infixowl.**EnumeratedClass**(*identifier=None*, *members=None*, *graph=None*)

Bases: *rdflib.extras.infixowl.OWLRDFListProxy*, *rdflib.extras.infixowl.Class*

__init__(*identifier=None*, *members=None*, *graph=None*)

__module__ = 'rdflib.extras.infixowl'

__repr__()

Returns the Manchester Syntax equivalent for this class

isPrimitive()

serialize(*graph*)

rdflib.extras.infixowl.**generateQName**(*graph*, *uri*)

rdflib.extras.infixowl.**GetIdentifiedClasses**(*graph*)

class rdflib.extras.infixowl.**Individual**(*identifier=None*, *graph=None*)

Bases: object

A typed individual

__init__(*identifier=None*, *graph=None*)

__module__ = 'rdflib.extras.infixowl'

clearInDegree()

clearOutDegree()

delete()

factoryGraph = <Graph identifier=N5accc83dfa8b46c0812c499f19cd921e (<class 'rdflib.graph.Graph'>)>

identifier

replace(*other*)

sameAs

serialize(*graph*)

type

exception rdflib.extras.infixowl.**MalformedClass**(*msg*)
Bases: exceptions.Exception

__init__(*msg*)

__module__ = 'rdflib.extras.infixowl'

__repr__()

rdflib.extras.infixowl.**manchesterSyntax**(*thing*, *store*, *boolean=None*, *transientList=False*)
Core serialization

class rdflib.extras.infixowl.**Ontology**(*identifier=None*, *imports=None*, *comment=None*, *graph=None*)
Bases: *rdflib.extras.infixowl.AnnotatableTerms*

The owl ontology metadata

__init__(*identifier=None*, *imports=None*, *comment=None*, *graph=None*)

__module__ = 'rdflib.extras.infixowl'

imports

setVersion(*version*)

class rdflib.extras.infixowl.**OWLRDFListProxy**(*rdfList*, *members=None*, *graph=None*)
Bases: object

__contains__(*item*)

__delitem__(*key*)

__eq__(*other*)
Equivalence of boolean class constructors is determined by equivalence of its members

__getitem__(*key*)

__iadd__(*other*)

__init__(*rdfList*, *members=None*, *graph=None*)

__iter__()

__len__()

__module__ = 'rdflib.extras.infixowl'

__setitem__(*key*, *value*)

append(*item*)

clear()

index(*item*)

class rdflib.extras.infixowl.**Property**(*identifier=None*, *graph=None*, *baseType=rdflib.term.URIRef(u'http://www.w3.org/2002/07/owl#ObjectProperty')*, *subPropertyOf=None*, *domain=None*, *range=None*, *inverseOf=None*, *otherType=None*, *equivalentProperty=None*, *comment=None*, *verbAnnotations=None*, *nameAnnotation=None*, *nameIsLabel=False*)

Bases: *rdflib.extras.infixowl.AnnotatableTerms*

axiom ::= 'DatatypeProperty(' datavaluedPropertyID ['Deprecated']

{ annotation } { 'super(' datavaluedPropertyID ')'} ['Functional'] { 'domain(' description ')' } { 'range(' dataRange ')' } ')'

'ObjectProperty(' individualvaluedPropertyID ['Deprecated'] { annotation } { 'super(' individualvaluedPropertyID ')' } ['inverseOf(' individualvaluedPropertyID ')'] ['Symmetric'] ['Functional' | 'InverseFunctional' | 'Functional' 'InverseFunctional' | 'Transitive'] { 'domain(' description ')' } { 'range(' description ')' } ')

__init__(*identifier=None*, *graph=None*, *baseType=rdflib.term.URIRef(u'http://www.w3.org/2002/07/owl#ObjectProperty')*, *subPropertyOf=None*, *domain=None*, *range=None*, *inverseOf=None*, *otherType=None*, *equivalentProperty=None*, *comment=None*, *verbAnnotations=None*, *nameAnnotation=None*, *nameIsLabel=False*)

__module__ = 'rdflib.extras.infixowl'

__repr__()

domain

extent

inverseOf

range

replace(*other*)

serialize(*graph*)

setupVerbAnnotations(*verbAnnotations*)

subPropertyOf

rdflib.extras.infixowl.**propertyOrIdentifier**(*thing*)

class rdflib.extras.infixowl.**Restriction**(*onProperty*, *graph=<Graph identifier=N6d52d90a4da949f6a01933c0e82c238d (<class 'rdflib.graph.Graph'>)>*, *allValuesFrom=None*, *someValuesFrom=None*, *value=None*, *cardinality=None*, *maxCardinality=None*, *minCardinality=None*, *identifier=None*)

Bases: *rdflib.extras.infixowl.Class*

restriction ::= 'restriction('

datavaluedPropertyID dataRestrictionComponent { dataRestrictionComponent } ')'

'restriction(' individualvaluedPropertyID individualRestrictionComponent { individualRestrictionComponent } ')'

__eq__(*other*)
Equivalence of restrictions is determined by equivalence of the property in question and the restriction 'range'

__hash__()

__init__(*onProperty*, *graph=<Graph identifier=N6d52d90a4da949f6a01933c0e82c238d (<class 'rdflib.graph.Graph'>)>*, *allValuesFrom=None*, *someValuesFrom=None*, *value=None*, *cardinality=None*, *maxCardinality=None*, *minCardinality=None*, *identifier=None*)

__module__ = 'rdflib.extras.infixowl'

__repr__()
Returns the Manchester Syntax equivalent for this restriction

allValuesFrom

cardinality

hasValue

isPrimitive()

maxCardinality

minCardinality

onProperty

restrictionKind()

restrictionKinds = [rdflib.term.URIRef(u'http://www.w3.org/2002/07/owl#allValuesFrom'), rdflib.term.URIRef(u'h

serialize(*graph*)

```
>>> g1 = Graph()
>>> g2 = Graph()
>>> EX = Namespace("http://example.com/")
>>> namespace_manager = NamespaceManager(g1)
>>> namespace_manager.bind('ex', EX, override=False)
>>> namespace_manager = NamespaceManager(g2)
>>> namespace_manager.bind('ex', EX, override=False)
>>> Individual.factoryGraph = g1
>>> prop = Property(EX.someProp, baseType=OWL_NS.DatatypeProperty)
>>> restr1 = (Property(
...     EX.someProp,
...     baseType=OWL_NS.DatatypeProperty)) | some | (Class(EX.Foo))
>>> restr1
( ex:someProp SOME ex:Foo )
>>> restr1.serialize(g2)
>>> Individual.factoryGraph = g2
>>> list(Property(
...      EX.someProp,baseType=None).type
... )
[rdflib.term.URIRef(
    u'http://www.w3.org/2002/07/owl#DatatypeProperty')]
```

someValuesFrom

rdflib.extras.infixowl.**termDeletionDecorator**(*prop*)

plugins Package

plugins Package Default plugins for rdflib.

This is a namespace package and contains the default plugins for rdflib.

memory Module

class rdflib.plugins.memory.**Memory**(*configuration=None*, *identifier=None*)

Bases: *rdflib.store.Store*

An in memory implementation of a triple store.

This triple store uses nested dictionaries to store triples. Each triple is stored in two such indices as follows spo[s][p][o] = 1 and pos[p][o][s] = 1.

Authors: Michel Pelletier, Daniel Krech, Stefan Niederhauser

__init__(*configuration=None*, *identifier=None*)

__len__(*context=None*)

__module__ = 'rdflib.plugins.memory'

add(*(subject*, *predicate*, *object)*, *context*, *quoted=False*)

Add a triple to the store of triples.

bind(*prefix*, *namespace*)

namespace(*prefix*)

namespaces()

prefix(*namespace*)

remove(*(subject*, *predicate*, *object)*, *context=None*)

triples(*(subject*, *predicate*, *object)*, *context=None*)

A generator over all the triples matching

class rdflib.plugins.memory.**IOMemory**(*configuration=None*, *identifier=None*)

Bases: *rdflib.store.Store*

An integer-key-optimized context-aware in-memory store.

Uses three dict indices (for subjects, objects and predicates) holding sets of triples. Context information is tracked in a separate dict, with the triple as key and a dict of {context: quoted} items as value. The context information is used to filter triple query results.

Memory usage is low due to several optimizations. RDF nodes are not stored directly in the indices; instead, the indices hold integer keys and the actual nodes are only stored once in int-to-object and object-to-int mapping dictionaries. A default context is determined based on the first triple that is added to the store, and no context information is actually stored for subsequent other triples with the same context information.

Most operations should be quite fast, but a triples() query with two bound parts requires a set intersection operation, which may be slow in some cases. When multiple contexts are used in the same store, filtering based on context has to be done after each query, which may also be slow.

__init__(*configuration=None*, *identifier=None*)

__len__(*context=None*)

__module__ = 'rdflib.plugins.memory'

add(*triple*, *context*, *quoted=False*)

add_graph(*graph*)

bind(*prefix*, *namespace*)

context_aware = True

contexts(*triple=None*)

formula_aware = True

graph_aware = True

namespace(*prefix*)

namespaces()

prefix(*namespace*)

remove(*triplepat*, *context=None*)

remove_graph(*graph*)

triples(*triplein*, *context=None*)

sleepycat Module

class rdflib.plugins.sleepycat.**Sleepycat**(*configuration=None*, *identifier=None*)

Bases: *rdflib.store.Store*

__init__(*configuration=None*, *identifier=None*)

__len__(*context=None*)

__module__ = 'rdflib.plugins.sleepycat'

add(*triple*, *context*, *quoted=False*, *txn=None*)

Add a triple to the store of triples.

add_graph(*graph*)

bind(*prefix*, *namespace*)

close(*commit_pending_transaction=False*)

context_aware = True

contexts(*triple=None*)

db_env = None

formula_aware = True

graph_aware = True

identifier

is_open()

namespace(*prefix*)

namespaces()

open(*path*, *create=True*)

prefix(*namespace*)

remove(*(subject*, *predicate*, *object)*, *context*, *txn=None*)

remove_graph(*graph*)

sync()

transaction_aware = False

triples((*subject*, *predicate*, *object*), *context=None*, *txn=None*)
A generator over all the triples matching

Subpackages

parsers Package

parsers Package

hturtle Module Extraction parser RDF embedded verbatim into HTML or XML files. This is based on:

- **The specification on embedding turtle into html:** http://www.w3.org/TR/turtle/#in-html

For SVG (and currently SVG only) the method also extracts an embedded RDF/XML data, per SVG specification

License: W3C Software License, http://www.w3.org/Consortium/Legal/copyright-software Author: Ivan Herman Copyright: W3C

class rdflib.plugins.parsers.hturtle.**HTurtle**(*options=None*, *base=''*, *media_type=''*)
Bases: *rdflib.plugins.parsers.pyRdfa.pyRdfa*

Bastardizing the RDFa 1.1 parser to do a hturtle extractions

__init__(*options=None*, *base=''*, *media_type=''*)

__module__ = 'rdflib.plugins.parsers.hturtle'

graph_from_DOM(*dom*, *graph*, *pgraph=None*)
Stealing the parsing function from the original class, to do turtle extraction only

class rdflib.plugins.parsers.hturtle.**HTurtleParser**
Bases: *rdflib.parser.Parser*

__module__ = 'rdflib.plugins.parsers.hturtle'

parse(*source*, *graph*, *pgraph=None*, *media_type=''*)
@param source: one of the input sources that the RDFLib package defined @type source: InputSource class instance @param graph: target graph for the triples; output graph, in RDFa spec. parlance @type graph: RDFLib Graph @keyword media_type: explicit setting of the preferred media type (a.k.a. content type) of the the RDFa source. None means the content type of the HTTP result is used, or a guess is made based on the suffix of a file @type media_type: string

notation3 Module notation3.py - Standalone Notation3 Parser Derived from CWM, the Closed World Machine

Authors of the original suite:

- Dan Connolly <@@>
- Tim Berners-Lee <@@>
- Yosi Scharf <@@>
- Joseph M. Reagle Jr. <reagle@w3.org>
- Rich Salz <rsalz@zolera.com>

http://www.w3.org/2000/10/swap/notation3.py

Copyright 2000-2007, World Wide Web Consortium. Copyright 2001, MIT. Copyright 2001, Zolera Systems Inc.

License: W3C Software License http://www.w3.org/Consortium/Legal/copyright-software

Modified by Sean B. Palmer Copyright 2007, Sean B. Palmer.

Modified to work with rdflib by Gunnar Aastrand Grimnes Copyright 2010, Gunnar A. Grimnes

exception rdflib.plugins.parsers.notation3.**BadSyntax**(*uri*, *lines*, *argstr*, *i*, *why*)
Bases: exceptions.SyntaxError

__init__(*uri*, *lines*, *argstr*, *i*, *why*)

__module__ = 'rdflib.plugins.parsers.notation3'

__str__()

__weakref__
list of weak references to the object (if defined)

message

class rdflib.plugins.parsers.notation3.**N3Parser**
Bases: *rdflib.plugins.parsers.notation3.TurtleParser*

An RDFLib parser for Notation3

See http://www.w3.org/DesignIssues/Notation3.html

__init__()

__module__ = 'rdflib.plugins.parsers.notation3'

parse(*source*, *graph*, *encoding='utf-8'*)

class rdflib.plugins.parsers.notation3.**TurtleParser**
Bases: *rdflib.parser.Parser*

An RDFLib parser for Turtle

See http://www.w3.org/TR/turtle/

__init__()

__module__ = 'rdflib.plugins.parsers.notation3'

parse(*source*, *graph*, *encoding='utf-8'*, *turtle=True*)

rdflib.plugins.parsers.notation3.**splitFragP**(*uriref*, *punct=0*)
split a URI reference before the fragment

Punctuation is kept.

e.g.

```
>>> splitFragP("abc#def")
('abc', '#def')
```

```
>>> splitFragP("abcdef")
('abcdef', '')
```

rdflib.plugins.parsers.notation3.**join**(*here*, *there*)
join an absolute URI and URI reference (non-ascii characters are supported/doctested; haven't checked the details of the IRI spec though)

`here` is assumed to be absolute. `there` is URI reference.

```
    >>> join('http://example/x/y/z', '../abc')
    'http://example/x/abc'
```

Raise ValueError if there uses relative path syntax but here has no hierarchical path.

```
    >>> join('mid:foo@example', '../foo')
    Traceback (most recent call last):
        raise ValueError(here)
    ValueError: Base <mid:foo@example> has no slash
    after colon - with relative '../foo'.
```

```
    >>> join('http://example/x/y/z', '')
    'http://example/x/y/z'
```

```
    >>> join('mid:foo@example', '#foo')
    'mid:foo@example#foo'
```

We grok IRIs

```
    >>> len(u'Andr\xe9')
    5
```

```
    >>> join('http://example.org/', u'#Andr\xe9')
    u'http://example.org/#Andr\xe9'
```

rdflib.plugins.parsers.notation3.**base**()
The base URI for this process - the Web equiv of cwd

Relative or abolute unix-standard filenames parsed relative to this yeild the URI of the file. If we had a reliable way of getting a computer name, we should put it in the hostname just to prevent ambiguity

rdflib.plugins.parsers.notation3.**runNamespace**()
Return a URI suitable as a namespace for run-local objects

rdflib.plugins.parsers.notation3.**uniqueURI**()
A unique URI

rdflib.plugins.parsers.notation3.**hexify**(*ustr*)
Use URL encoding to return an ASCII string corresponding to the given UTF8 string

```
    >>> hexify("http://example/a b")
    'http://example/a%20b'
```

nquads Module This is a rdflib plugin for parsing NQuad files into Conjunctive graphs that can be used and queried. The store that backs the graph *must* be able to handle contexts.

```
>>> from rdflib import ConjunctiveGraph, URIRef, Namespace
>>> g = ConjunctiveGraph()
>>> data = open("test/nquads.rdflib/example.nquads", "rb")
>>> g.parse(data, format="nquads")
<Graph identifier=... (<class 'rdflib.graph.Graph'>)>
>>> assert len(g.store) == 449
>>> # There should be 16 separate contexts
>>> assert len([x for x in g.store.contexts()]) == 16
>>> # is the name of entity E10009 "Arco Publications"?
>>> #   (in graph http://bibliographica.org/entity/E10009)
>>> # Looking for:
>>> # <http://bibliographica.org/entity/E10009>
>>> #   <http://xmlns.com/foaf/0.1/name>
```

```
>>> #   "Arco Publications"
>>> #   <http://bibliographica.org/entity/E10009>
>>> s = URIRef("http://bibliographica.org/entity/E10009")
>>> FOAF = Namespace("http://xmlns.com/foaf/0.1/")
>>> assert(g.value(s, FOAF.name).eq("Arco Publications"))
```

class rdflib.plugins.parsers.nquads.**NQuadsParser**(*sink=None*)
Bases: *rdflib.plugins.parsers.ntriples.NTriplesParser*

__module__ = 'rdflib.plugins.parsers.nquads'

parse(*inputsource*, *sink*, ***kwargs*)
Parse f as an N-Triples file.

parseline()

nt Module

class rdflib.plugins.parsers.nt.**NTSink**(*graph*)
Bases: object

__init__(*graph*)

__module__ = 'rdflib.plugins.parsers.nt'

triple(*s*, *p*, *o*)

class rdflib.plugins.parsers.nt.**NTParser**
Bases: *rdflib.parser.Parser*

parser for the ntriples format, often stored with the .nt extension

See http://www.w3.org/TR/rdf-testcases/#ntriples

__init__()

__module__ = 'rdflib.plugins.parsers.nt'

parse(*source*, *sink*, *baseURI=None*)

ntriples Module N-Triples Parser License: GPL 2, W3C, BSD, or MIT Author: Sean B. Palmer, inamidst.com

rdflib.plugins.parsers.ntriples.**unquote**(*s*)
Unquote an N-Triples string.

rdflib.plugins.parsers.ntriples.**uriquote**(*uri*)

class rdflib.plugins.parsers.ntriples.**Sink**
Bases: object

__init__()

__module__ = 'rdflib.plugins.parsers.ntriples'

triple(*s*, *p*, *o*)

class rdflib.plugins.parsers.ntriples.**NTriplesParser**(*sink=None*)
Bases: *object*

An N-Triples Parser.

Usage:

```
p = NTriplesParser(sink=MySink())
sink = p.parse(f) # file; use parsestring for a string
```

__init__(*sink=None*)

__module__ = 'rdflib.plugins.parsers.ntriples'

eat(*pattern*)

literal()

nodeid()

object()

parse(*f*)
Parse f as an N-Triples file.

parseline()

parsestring(*s*)
Parse s as an N-Triples string.

peek(*token*)

predicate()

readline()
Read an N-Triples line from buffered input.

subject()

uriref()

rdfxml Module An RDF/XML parser for RDFLib

rdflib.plugins.parsers.rdfxml.**create_parser**(*target*, *store*)

class rdflib.plugins.parsers.rdfxml.**BagID**(*val*)
Bases: *rdflib.term.URIRef*

__init__(*val*)

__module__ = 'rdflib.plugins.parsers.rdfxml'

__slots__ = ['li']

li

next_li()

class rdflib.plugins.parsers.rdfxml.**ElementHandler**
Bases: *object*

__init__()

__module__ = 'rdflib.plugins.parsers.rdfxml'

__slots__ = ['start', 'char', 'end', 'li', 'id', 'base', 'subject', 'predicate', 'object', 'list', 'language', 'datatype', 'declared

base

char

data

datatype

declared

end

id

language

li

list

next_li()

object

predicate

start

subject

class rdflib.plugins.parsers.rdfxml.**RDFXMLHandler**(*store*)
Bases: xml.sax.handler.ContentHandler

__init__(*store*)

__module__ = 'rdflib.plugins.parsers.rdfxml'

absolutize(*uri*)

add_reified(*sid*, *(s*, *p*, *o)*)

characters(*content*)

convert(*name*, *qname*, *attrs*)

current

document_element_start(*name*, *qname*, *attrs*)

endElementNS(*name*, *qname*)

endPrefixMapping(*prefix*)

error(*message*)

get_current()

get_next()

get_parent()

ignorableWhitespace(*content*)

list_node_element_end(*name*, *qname*)

literal_element_char(*data*)

literal_element_end(*name*, *qname*)

literal_element_start(*name*, *qname*, *attrs*)

next

node_element_end(*name*, *qname*)

node_element_start(*name*, *qname*, *attrs*)

parent

processingInstruction(*target*, *data*)

property_element_char(*data*)

property_element_end(*name*, *qname*)

property_element_start(*name*, *qname*, *attrs*)

reset()

setDocumentLocator(*locator*)

startDocument()

startElementNS(*name*, *qname*, *attrs*)

startPrefixMapping(*prefix*, *namespace*)

class rdflib.plugins.parsers.rdfxml.**RDFXMLParser**
Bases: *rdflib.parser.Parser*

__init__()

__module__ = 'rdflib.plugins.parsers.rdfxml'

parse(*source*, *sink*, ***args*)

structureddata Module Extraction parsers for structured data embedded into HTML or XML files. The former may include RDFa or microdata. The syntax and the extraction procedures are based on:

- The RDFa specifications: http://www.w3.org/TR/#tr_RDFa
- The microdata specification: http://www.w3.org/TR/microdata/
- The specification of the microdata to RDF conversion:

http://www.w3.org/TR/microdata-rdf/

License: W3C Software License, http://www.w3.org/Consortium/Legal/copyright-software Author: Ivan Herman Copyright: W3C

class rdflib.plugins.parsers.structureddata.**MicrodataParser**
Bases: *rdflib.parser.Parser*

Wrapper around an HTML5 microdata, extracted and converted into RDF. For the specification of microdata, see the relevant section of the HTML5 spec: http://www.w3.org/TR/microdata/; for the algorithm used to extract microdata into RDF, see http://www.w3.org/TR/microdata-rdf/.

__module__ = 'rdflib.plugins.parsers.structureddata'

parse(*source*, *graph*, *vocab_expansion=False*, *vocab_cache=False*)
@param source: one of the input sources that the RDFLib package defined @type source: InputSource class instance @param graph: target graph for the triples; output graph, in RDFa spec. parlance @type graph: RDFLib Graph @keyword vocab_expansion: whether the RDFa @vocab attribute should also mean vocabulary expansion (see the RDFa 1.1 spec for further

details)

@type vocab_expansion: Boolean @keyword vocab_cache: in case vocab expansion is used, whether the expansion data (i.e., vocabulary) should be cached locally. This requires the ability for the local application to write on the local file system @type vocab_chache: Boolean @keyword rdfOutput: whether Exceptions should be catched and added, as triples, to the processor graph, or whether they should be raised. @type rdfOutput: Boolean

class rdflib.plugins.parsers.structureddata.**RDFa10Parser**
Bases: *rdflib.parser.Parser*

This is just a convenience class to wrap around the RDFa 1.0 parser.

__module__ = 'rdflib.plugins.parsers.structureddata'

parse(*source*, *graph*, *pgraph=None*, *media_type=''*)

@param source: one of the input sources that the RDFLib package defined @type source: InputSource class instance @param graph: target graph for the triples; output graph, in RDFa spec. parlance @type graph: RDFLib Graph @keyword pgraph: target for error and warning triples; processor graph, in RDFa spec. parlance. If set to None, these triples are ignored @type pgraph: RDFLib Graph @keyword media_type: explicit setting of the preferred media type (a.k.a. content type) of the the RDFa source. None means the content type of the HTTP result is used, or a guess is made based on the suffix of a file @type media_type: string @keyword rdfOutput: whether Exceptions should be catched and added, as triples, to the processor graph, or whether they should be raised. @type rdfOutput: Boolean

class rdflib.plugins.parsers.structureddata.**RDFaParser**

Bases: *rdflib.parser.Parser*

Wrapper around the RDFa 1.1 parser. For further details on the RDFa 1.1 processing, see the relevant W3C documents at http://www.w3.org/TR/#tr_RDFa. RDFa 1.1 is defined for XHTML, HTML5, SVG and, in general, for any XML language.

Note that the parser can also handle RDFa 1.0 if the extra parameter is used and/or the input source uses RDFa 1.0 specific @version or DTD-s.

__module__ = 'rdflib.plugins.parsers.structureddata'

parse(*source*, *graph*, *pgraph=None*, *media_type=''*, *rdfa_version=None*, *embedded_rdf=False*, *space_preserve=True*, *vocab_expansion=False*, *vocab_cache=False*, *refresh_vocab_cache=False*, *vocab_cache_report=False*, *check_lite=False*)

@param source: one of the input sources that the RDFLib package defined @type source: InputSource class instance @param graph: target graph for the triples; output graph, in RDFa spec. parlance @type graph: RDFLib Graph @keyword pgraph: target for error and warning triples; processor graph, in RDFa spec. parlance. If set to None, these triples are ignored @type pgraph: RDFLib Graph @keyword media_type: explicit setting of the preferred media type (a.k.a. content type) of the the RDFa source. None means the content type of the HTTP result is used, or a guess is made based on the suffix of a file @type media_type: string @keyword rdfa_version: 1.0 or 1.1. If the value is "", then, by default, 1.1 is used unless the source has explicit signals to use 1.0 (e.g., using a @version attribute, using a DTD set up for 1.0, etc) @type rdfa_version: string @keyword embedded_rdf: some formats allow embedding RDF in other formats: (X)HTML can contain turtle in a special <script> element, SVG can have RDF/XML embedded in a <metadata> element. This flag controls whether those triples should be interpreted and added to the output graph. Some languages (e.g., SVG) require this, and the flag is ignored. @type embedded_rdf: Boolean @keyword space_preserve: by default, space in the HTML source must be preserved in the generated literal; this behavior can be switched off @type space_preserve: Boolean @keyword vocab_expansion: whether the RDFa @vocab attribute should also mean vocabulary expansion (see the RDFa 1.1 spec for further details) @type vocab_expansion: Boolean @keyword vocab_cache: in case vocab expansion is used, whether the expansion data (i.e., vocabulary) should be cached locally. This requires the ability for the local application to write on the local file system @type vocab_chache: Boolean @keyword vocab_cache_report: whether the details of vocabulary file caching process should be reported in the processor graph as information (mainly useful for debug) @type vocab_cache_report: Boolean @keyword refresh_vocab_cache: whether the caching checks of vocabs should be by-passed, ie, if caches should be re-generated regardless of the stored date (important for vocab development) @type refresh_vocab_cache: Boolean @keyword check_lite: generate extra warnings in case the input source is not RDFa 1.1 check_lite @type check_lite: Boolean

class rdflib.plugins.parsers.structureddata.**StructuredDataParser**

Bases: *rdflib.parser.Parser*

Convenience parser to extract both RDFa (including embedded Turtle) and microdata from an HTML file. It is simply a wrapper around the specific parsers.

__module__ = 'rdflib.plugins.parsers.structureddata'

parse(*source*, *graph*, *pgraph=None*, *rdfa_version=''*, *vocab_expansion=False*, *vocab_cache=False*, *media_type='text/html'*)

@param source: one of the input sources that the RDFLib package defined @type source: InputSource class instance @param graph: target graph for the triples; output graph, in RDFa spec. parlance @keyword rdfa_version: 1.0 or 1.1. If the value is "", then, by default, 1.1 is used unless the source has explicit signals to use 1.0 (e.g., using a @version attribute, using a DTD set up for 1.0, etc) @type rdfa_version: string @type graph: RDFLib Graph @keyword pgraph: target for error and warning triples; processor graph, in RDFa spec. parlance. If set to None, these triples are ignored @type pgraph: RDFLib Graph @keyword vocab_expansion: whether the RDFa @vocab attribute should also mean vocabulary expansion (see the RDFa 1.1 spec for further

details)

@type vocab_expansion: Boolean @keyword vocab_cache: in case vocab expansion is used, whether the expansion data (i.e., vocabulary) should be cached locally. This requires the ability for the local application to write on the local file system @type vocab_chache: Boolean @keyword rdfOutput: whether Exceptions should be catched and added, as triples, to the processor graph, or whether they should be raised. @type rdfOutput: Boolean

trix Module A TriX parser for RDFLib

rdflib.plugins.parsers.trix.**create_parser**(*store*)

class rdflib.plugins.parsers.trix.**TriXHandler**(*store*)

Bases: xml.sax.handler.ContentHandler

An Sax Handler for TriX. See http://sw.nokia.com/trix/

__init__(*store*)

__module__ = 'rdflib.plugins.parsers.trix'

characters(*content*)

endElementNS(*name*, *qname*)

endPrefixMapping(*prefix*)

error(*message*)

get_bnode(*label*)

ignorableWhitespace(*content*)

processingInstruction(*target*, *data*)

reset()

setDocumentLocator(*locator*)

startDocument()

startElementNS(*name*, *qname*, *attrs*)

startPrefixMapping(*prefix*, *namespace*)

class rdflib.plugins.parsers.trix.**TriXParser**

Bases: *rdflib.parser.Parser*

A parser for TriX. See http://sw.nokia.com/trix/

__init__()

__module__ = 'rdflib.plugins.parsers.trix'

parse(*source*, *sink*, ***args*)

Subpackages

pyMicrodata Package

pyMicrodata Package This module implements the microdata->RDF algorithm, as documented by the U{W3C Semantic Web Interest Group Note<http://www.w3.org/TR/2012/NOTE-microdata-rdf-20120308/>}.

The module can be used via a stand-alone script (an example is part of the distribution) or bound to a CGI script as a Web Service. An example CGI script is also added to the distribution. Both the local script and the distribution may have to be adapted to local circumstances.

(Simple) Usage

From a Python file, expecting a Turtle output:: from pyMicrodata import pyMicrodata print pyMicrodata().rdf_from_source('filename')

Other output formats are also possible. E.g., to produce RDF/XML output, one could use:: from pyMicrodata import pyMicrodata print pyMicrodata().rdf_from_source('filename', outputFormat='pretty-xml')

It is also possible to embed an RDFa processing. Eg, using:: from pyMicrodata import pyMicrodata graph = pyMicrodata().graph_from_source('filename')

returns an RDFLib.Graph object instead of a serialization thereof. See the the description of the L{pyMicrodata class<pyMicrodata.pyMicrodata>} for further possible entry points details.

There is also, as part of this module, a L{separate entry for CGI calls<processURI>}.

Return formats By default, the output format for the graph is RDF/XML. At present, the following formats are also available (with the corresponding key to be used in the package entry points):

- "xml": U{RDF/XML<http://www.w3.org/TR/rdf-syntax-grammar/>}
- "turtle": U{Turtle<http://www.w3.org/TR/turtle/>} (default)
- "nt": U{N-triple<http://www.w3.org/TR/rdf-testcases/#ntriples>}
- "json": U{JSON-LD<http://json-ld.org/spec/latest/json-ld-syntax/>}

@summary: Microdata parser (distiller) @requires: Python version 2.5 or up @requires: U{RDFLib<http://rdflib.net>} @requires: U{html5lib<http://code.google.com/p/html5lib/>} for the HTML5 parsing; note possible dependecies on Python's version on the project's web site @organization: U{World Wide Web Consortium<http://www.w3.org>} @author: U{Ivan Herman<http://www.w3.org/People/Ivan/>} @license: This software is available for use under the U{W3C® SOFTWARE NOTICE AND LICENSE<href="http://www.w3.org/Consortium/Legal/2002/copyright-software-20021231">} @copyright: W3C

exception rdflib.plugins.parsers.pyMicrodata.**HTTPError**(*http_msg*, *http_code*)
Bases: *rdflib.plugins.parsers.pyMicrodata.MicrodataError*

Raised when HTTP problems are detected. It does not add any new functionality to the Exception class.

__init__(*http_msg*, *http_code*)

__module__ = 'rdflib.plugins.parsers.pyMicrodata'

exception rdflib.plugins.parsers.pyMicrodata.**MicrodataError**(*msg*)
Bases: exceptions.Exception

Superclass exceptions representing error conditions defined by the RDFa 1.1 specification. It does not add any new functionality to the Exception class.

__init__(*msg*)

__module__ = 'rdflib.plugins.parsers.pyMicrodata'

__weakref__
list of weak references to the object (if defined)

rdflib.plugins.parsers.pyMicrodata.**processURI**(*uri*, *outputFormat*, *form*)
The standard processing of a microdata uri options in a form, ie, as an entry point from a CGI call.

The call accepts extra form options (eg, HTTP GET options) as follows:

@param uri: URI to access. Note that the "text:" and "uploaded:" values are treated separately; the former is for textual intput (in which case a StringIO is used to get the data) and the latter is for uploaded file, where the form gives access to the file directly. @param outputFormat: serialization formats, as understood by RDFLib. Note that though "turtle" is a possible parameter value, some versions of the RDFLib turtle generation does funny (though legal) things with namespaces, defining unusual and unwanted prefixes... @param form: extra call options (from the CGI call) to set up the local options (if any) @type form: cgi FieldStorage instance @return: serialized graph @rtype: string

class rdflib.plugins.parsers.pyMicrodata.**pyMicrodata**(*base=''*, *vocab_expansion=False*, *vocab_cache=True*)
Main processing class for the distiller @ivar base: the base value for processing @ivar http_status: HTTP Status, to be returned when the package is used via a CGI entry. Initially set to 200, may be modified by exception handlers

__init__(*base=''*, *vocab_expansion=False*, *vocab_cache=True*)
@keyword base: URI for the default "base" value (usually the URI of the file to be processed) @keyword vocab_expansion: whether vocab expansion should be performed or not @type vocab_expansion: Boolean @keyword vocab_cache: if vocabulary expansion is done, then perform caching of the vocabulary data @type vocab_cache: Boolean

__module__ = 'rdflib.plugins.parsers.pyMicrodata'

graph_from_DOM(*dom*, *graph=None*)
Extract the RDF Graph from a DOM tree. @param dom: a DOM Node element, the top level entry node for the whole tree (to make it clear, a dom.documentElement is used to initiate processing) @keyword graph: an RDF Graph (if None, than a new one is created) @type graph: rdflib Graph instance. If None, a new one is created. @return: an RDF Graph @rtype: rdflib Graph instance

graph_from_source(*name*, *graph=None*, *rdfOutput=False*)
Extract an RDF graph from an microdata source. The source is parsed, the RDF extracted, and the RDF Graph is returned. This is a front-end to the L{pyMicrodata.graph_from_DOM} method.

@param name: a URI, a file name, or a file-like object @return: an RDF Graph @rtype: rdflib Graph instance

rdf_from_source(*name*, *outputFormat='pretty-xml'*, *rdfOutput=False*)
Extract and RDF graph from an RDFa source and serialize it in one graph. The source is parsed, the RDF extracted, and serialization is done in the specified format. @param name: a URI, a file name, or a file-like object @keyword outputFormat: serialization format. Can be one of "turtle", "n3", "xml", "pretty-xml", "nt". "xml" and "pretty-xml", as well as "turtle" and "n3" are synonyms. @return: a serialized RDF Graph @rtype: string

rdf_from_sources(*names*, *outputFormat='pretty-xml'*, *rdfOutput=False*)
Extract and RDF graph from a list of RDFa sources and serialize them in one graph. The sources are parsed,

the RDF extracted, and serialization is done in the specified format. @param names: list of sources, each can be a URI, a file name, or a file-like object @keyword outputFormat: serialization format. Can be one of “turtle”, “n3”, “xml”, “pretty-xml”, “nt”. “xml” and “pretty-xml”, as well as “turtle” and “n3” are synonyms. @return: a serialized RDF Graph @rtype: string

microdata Module The core of the Microdata->RDF conversion, a more or less verbatim implementation of the U{W3C IG Note<http://www.w3.org/TR/microdata-rdf/>}. Because the implementation was also used to check the note itself, it tries to be fairly close to the text.

@organization: U{World Wide Web Consortium<http://www.w3.org>} @author: U{Ivan Herman<a href=”http://www.w3.org/People/Ivan/“>} @license: This software is available for use under the U{W3C® SOFTWARE NOTICE AND LICENSE<href=”http://www.w3.org/Consortium/Legal/2002/copyright-software-20021231“>}

class rdflib.plugins.parsers.pyMicrodata.microdata.**Evaluation_Context**

Evaluation context structure. See Section 4.1 of the U{W3C IG Note<http://www.w3.org/TR/microdata-rdf/>}for the details.

@ivar current_type : an absolute URL for the current type, used when an item does not contain an item type @ivar memory: mapping from items to RDF subjects @type memory: dictionary @ivar current_name: an absolute URL for the in-scope name, used for generating URIs for properties of items without an item type @ivar current_vocabulary: an absolute URL for the current vocabulary, from the registry

__init__()

__module__ = ‘rdflib.plugins.parsers.pyMicrodata.microdata’

__str__()

get_memory(*item*)

Get the memory content (ie, RDF subject) for ‘item’, or None if not stored yet @param item: an ‘item’, in microdata terminology @type item: DOM Element Node @return: None, or an RDF Subject (URIRef or BNode)

new_copy(*itype*)

During the generation algorithm a new copy of the current context has to be done with a new current type.

At the moment, the content of memory is copied, ie, a fresh dictionary is created and the content copied over. Not clear whether that is necessary, though, maybe a simple reference is enough... @param itype : an absolute URL for the current type @return: a new evaluation context instance

set_memory(*item*, *subject*)

Set the memory content, ie, the subject, for ‘item’. @param item: an ‘item’, in microdata terminology @type item: DOM Element Node @param subject: RDF Subject @type subject: URIRef or Blank Node

class rdflib.plugins.parsers.pyMicrodata.microdata.**Microdata**(*document*, *base=None*)

This class encapsulates methods that are defined by the U{microdata spec<http://dev.w3.org/html5/md/Overview.html>}, as opposed to the RDF conversion note.

@ivar document: top of the DOM tree, as returned by the HTML5 parser @ivar base: the base URI of the Dom tree, either set from the outside or via a @base element

__init__(*document*, *base=None*)

@param document: top of the DOM tree, as returned by the HTML5 parser @param base: the base URI of the Dom tree, either set from the outside or via a @base element

__module__ = ‘rdflib.plugins.parsers.pyMicrodata.microdata’

getElementById(*id*)

This is a method defined for DOM 2 HTML, but the HTML5 parser does not seem to define it. Oh well...

@param id: value of an @id attribute to look for @return: array of nodes whose @id attribute matches C{id} (formally, there should be only one...)

get_item_properties(*item*)
Collect the item's properties, ie, all DOM descendent nodes with @itemprop until the subtree hits another @itemscope. @itemrefs are also added at this point.

@param item: current item @type item: DOM Node @return: array of items, ie, DOM Nodes

get_top_level_items()
A top level item is and element that has the @itemscope set, but no @itemtype. They have to be collected in pre-order and depth-first fashion.

@return: list of items (ie, DOM Nodes)

class rdflib.plugins.parsers.pyMicrodata.microdata.**MicrodataConversion**(*document*, *graph*, *base=None*, *vocab_expansion=False*, *vocab_cache=True*)
Bases: *rdflib.plugins.parsers.pyMicrodata.microdata.Microdata*

Top level class encapsulating the conversion algorithms as described in the W3C note.

@ivar graph: an RDF graph; an RDFLib Graph @type graph: RDFLib Graph @ivar document: top of the DOM tree, as returned by the HTML5 parser @ivar ns_md: the Namespace for the microdata vocabulary @ivar base: the base of the Dom tree, either set from the outside or via a @base element

__init__(*document*, *graph*, *base=None*, *vocab_expansion=False*, *vocab_cache=True*)
@param graph: an RDF graph; an RDFLib Graph @type graph: RDFLib Graph @param document: top of the DOM tree, as returned by the HTML5 parser @keyword base: the base of the Dom tree, either set from the outside or via a @base element @keyword vocab_expansion: whether vocab expansion should be performed or not @type vocab_expansion: Boolean @keyword vocab_cache: if vocabulary expansion is done, then perform caching of the vocabulary data @type vocab_cache: Boolean

__module__ = 'rdflib.plugins.parsers.pyMicrodata.microdata'

convert()
Top level entry to convert and generate all the triples. It finds the top level items, and generates triples for each of them; additionally, it generates a top level entry point to the items from base in the form of an RDF list.

generate_predicate_URI(*name*, *context*)
Generate a full URI for a predicate, using the type, the vocabulary, etc.

For details of this entry, see Section 4.4 @param name: name of the property, ie, what appears in @itemprop @param context: an instance of an evaluation context @type context: L{Evaluation_Context}

generate_property_values(*subject*, *predicate*, *objects*, *context*)
Generate the property values for a specific subject and predicate. The context should specify whether the objects should be added in an RDF list or each triples individually.

@param subject: RDF subject @type subject: RDFLib Node (URIRef or blank node) @param predicate: RDF predicate @type predicate: RDFLib URIRef @param objects: RDF objects @type objects: list of RDFLib nodes (URIRefs, Blank Nodes, or literals) @param context: evaluation context @type context: L{Evaluation_Context}

generate_triples(*item*, *context*)
Generate the triples for a specific item. See the W3C Note for the details.

@param item: the DOM Node for the specific item @type item: DOM Node @param context: an instance of an evaluation context @type context: L{Evaluation_Context} @return: a URIRef or a BNode for the (RDF) subject

get_property_value(*node*, *context*)
Generate an RDF object, ie, the value of a property. Note that if this element contains an @itemscope, then a recursive call to L{MicrodataConversion.generate_triples} is done and the return value of that method (ie, the subject for the corresponding item) is return as an object.

Otherwise, either URIRefs are created for <a>, <img>, etc, elements, or a Literal; the latter gets a time-related type for the <time> element.

@param node: the DOM Node for which the property values should be generated @type node: DOM Node @param context: an instance of an evaluation context @type context: L{Evaluation_Context} @return: an RDF resource (URIRef, BNode, or Literal)

class rdflib.plugins.parsers.pyMicrodata.microdata.**PropertySchemes**

__module__ = 'rdflib.plugins.parsers.pyMicrodata.microdata'

contextual = 'contextual'

vocabulary = 'vocabulary'

class rdflib.plugins.parsers.pyMicrodata.microdata.**ValueMethod**

__module__ = 'rdflib.plugins.parsers.pyMicrodata.microdata'

list = 'list'

unordered = 'unordered'

registry Module Hardcoded version of the current microdata->RDF registry. There is also a local registry to include some test cases. Finally, there is a local dictionary for prefix mapping for the registry items; these are the preferred prefixes for those vocabularies, and are used to make the output nicer.

@organization: U{World Wide Web Consortium<http://www.w3.org>} @author: U{Ivan Herman<a href="http://www.w3.org/People/Ivan/">} @license: This software is available for use under the U{W3C® SOFTWARE NOTICE AND LICENSE<href="http://www.w3.org/Consortium/Legal/2002/copyright-software-20021231">}

utils Module Various utilities for pyMicrodata

@organization: U{World Wide Web Consortium<http://www.w3.org>} @author: U{Ivan Herman<a href="http://www.w3.org/People/Ivan/">} @license: This software is available for use under the U{W3C® SOFTWARE NOTICE AND LICENSE<href="http://www.w3.org/Consortium/Legal/2002/copyright-software-20021231">}

class rdflib.plugins.parsers.pyMicrodata.utils.**URIOpener**(*name*)
A wrapper around the urllib2 method to open a resource. Beyond accessing the data itself, the class sets the content location. The class also adds an accept header to the outgoing request, namely text/html and application/xhtml+xml (unless set explicitly by the caller).

@ivar data: the real data, ie, a file-like object @ivar headers: the return headers as sent back by the server @ivar location: the real location of the data (ie, after possible redirection and content negotiation)

CONTENT_LOCATION = 'Content-Location'

__init__(*name*)
@param name: URL to be opened @keyword additional_headers: additional HTTP request headers to be added to the call

__module__ = 'rdflib.plugins.parsers.pyMicrodata.utils'

rdflib.plugins.parsers.pyMicrodata.utils.**fragment_escape**(*name*)

rdflib.plugins.parsers.pyMicrodata.utils.**generate_RDF_collection**(*graph*, *vals*)
Generate an RDF List from vals, returns the head of the list @param graph: RDF graph @type graph: RDFLib Graph @param vals: array of RDF Resources @return: head of the List (an RDF Resource)

rdflib.plugins.parsers.pyMicrodata.utils.**generate_URI**(*base*, *v*)
Generate an (absolute) URI; if val is a fragment, then using it with base, otherwise just return the value @param base: Absolute URI for base @param v: relative or absolute URI

rdflib.plugins.parsers.pyMicrodata.utils.**get_Literal**(*Pnode*)
Get (recursively) the full text from a DOM Node.

@param Pnode: DOM Node @return: string

rdflib.plugins.parsers.pyMicrodata.utils.**get_lang**(*node*)

rdflib.plugins.parsers.pyMicrodata.utils.**get_lang_from_hierarchy**(*document*, *node*)

rdflib.plugins.parsers.pyMicrodata.utils.**get_time_type**(*string*)
Check whether the string abides to one of the accepted time related datatypes, and returns that one if yes @param string: the attribute value to be checked @return : a datatype URI or None

rdflib.plugins.parsers.pyMicrodata.utils.**is_absolute_URI**(*uri*)

pyRdfa Package

pyRdfa Package RDFa 1.1 parser, also referred to as a "RDFa Distiller". It is deployed, via a CGI front-end, on the U{W3C RDFa 1.1 Distiller page<http://www.w3.org/2012/pyRdfa/>}.

For details on RDFa, the reader should consult the U{RDFa Core 1.1<http://www.w3.org/TR/rdfa-core/>}, U{XHTML+RDFa1.1<http://www.w3.org/TR/2010/xhtml-rdfa>}, and the U{RDFa 1.1 Lite<http://www.w3.org/TR/rdfa-lite/>} documents. The U{RDFa 1.1 Primer<http://www.w3.org/TR/owl2-primer/>} may also prove helpful.

This package can also be downloaded U{from GitHub<https://github.com/RDFLib/pyrdfa3>}. The distribution also includes the CGI front-end and a separate utility script to be run locally.

Note that this package is an updated version of a U{previous RDFa distiller<http://www.w3.org/2007/08/pyRdfa>} that was developed for RDFa 1.0. Although it reuses large portions of that code, it has been quite thoroughly rewritten, hence put in a completely different project. (The version numbering has been continued, though, to avoid any kind of misunderstandings. This version has version numbers "3.0.0" or higher.)

(Simple) Usage

From a Python file, expecting a Turtle output:: from pyRdfa import pyRdfa print pyRdfa().rdf_from_source('filename')

Other output formats are also possible. E.g., to produce RDF/XML output, one could use:: from pyRdfa import pyRdfa print pyRdfa().rdf_from_source('filename', outputFormat='pretty-xml')

It is also possible to embed an RDFa processing. Eg, using:: from pyRdfa import pyRdfa graph = pyRdfa().graph_from_source('filename')

returns an RDFLib.Graph object instead of a serialization thereof. See the the description of the L{pyRdfa class<pyRdfa.pyRdfa>} for further possible entry points details.

There is also, as part of this module, a L{separate entry for CGI calls<processURI>}.

Return (serialization) formats The package relies on RDFLib. By default, it relies therefore on the serializers coming with the local RDFLib distribution. However, there has been some issues with serializers of older RDFLib releases; also, some output formats, like JSON-LD, are not (yet) part of the standard RDFLib distribution. A companion package, called pyRdfaExtras, is part of the download, and it includes some of those extra serializers. The extra format (not part of the RDFLib core) is U{JSON-LD<http://json-ld.org/spec/latest/json-ld-syntax/>}, whose 'key' is 'json', when used in the 'parse' method of an RDFLib graph.

Options The package also implements some optional features that are not part of the RDFa recommendations. At the moment these are:

- possibility for plain literals to be normalized in terms of white spaces. Default: false. (The RDFa specification requires keeping the white spaces and leave applications to normalize them, if needed)
- inclusion of embedded RDF: Turtle content may be enclosed in a C{script} element and typed as C{text/turtle}, U{defined by the RDF Working Group<http://www.w3.org/TR/turtle/>}. Alternatively, some XML dialects (e.g., SVG) allows the usage of RDF/XML as part of their core content to define metadata in RDF. For both of these cases pyRdfa parses these serialized RDF content and adds the resulting triples to the output Graph. Default: true.
- extra, built-in transformers are executed on the DOM tree prior to RDFa processing (see below). These transformers can be provided by the end user.

Options are collected in an instance of the L{Options} class and may be passed to the processing functions as an extra argument
from pyRdfa.options import Options options = Options(embedded_rdf=True) print pyRdfa(options=options).rdf_from_source('filename')

See the description of the L{Options} class for the details.

Host Languages RDFa 1.1. Core is defined for generic XML; there are specific documents to describe how the generic specification is applied to XHTML and HTML5.

pyRdfa makes an automatic switch among these based on the content type of the source as returned by an HTTP request. The following are the possible host languages:

- if the content type is C{text/html}, the content is HTML5
- if the content type is C{application/xhtml+xml} I{and} the right DTD is used, the content is XHTML1
- if the content type is C{application/xhtml+xml} and no or an unknown DTD is used, the content is XHTML5
- if the content type is C{application/svg+xml}, the content type is SVG
- if the content type is C{application/atom+xml}, the content type is SVG
- if the content type is C{application/xml} or C{application/xxx+xml} (but 'xxx' is not 'atom' or 'svg'), the content type is XML

If local files are used, pyRdfa makes a guess on the content type based on the file name suffix: C{.html} is for HTML5, C{.xhtml} for XHTML1, C{.svg} for SVG, anything else is considered to be general XML. Finally, the content type may be set by the caller when initializing the L{pyRdfa class<pyRdfa.pyRdfa>}.

Beyond the differences described in the RDFa specification, the main difference is the parser used to parse the source. In the case of HTML5, pyRdfa uses an U{HTML5 parser<http://code.google.com/p/html5lib/>}; for all other cases the simple XML parser, part of the core Python environment, is used. This may be significant in the case of erronuous

sources: indeed, the HTML5 parser may do adjustments on the DOM tree before handing it over to the distiller. Furthermore, SVG is also recognized as a type that allows embedded RDF in the form of RDF/XML.

See the variables in the L{host} module if a new host language is added to the system. The current host language information is available for transformers via the option argument, too, and can be used to control the effect of the transformer.

Vocabularies RDFa 1.1 has the notion of vocabulary files (using the C{@vocab} attribute) that may be used to expand the generated RDF graph. Expansion is based on some very simply RDF Schema and OWL statements on sub-properties and sub-classes, and equivalences.

pyRdfa implements this feature, although it does not do this by default. The extra C{vocab_expansion} parameter should be us

from pyRdfa.options import Options options = Options(vocab_expansion=True) print pyRdfa(options=options).rdf_from_source('filename')

The triples in the vocabulary files themselves (i.e., the small ontology in RDF Schema and OWL) are removed from the result, leaving the inferred property and type relationships only (additionally to the "core" RDF content).

Vocabulary caching By default, pyRdfa uses a caching mechanism instead of fetching the vocabulary files each time their URI is met as a C{@vocab} attribute value. (This behavior can be switched off setting the C{vocab_cache} option to false.)

Caching happens in a file system directory. The directory itself is determined by the platform the tool is used on, namely:

- On Windows, it is the C{pyRdfa-cache} subdirectory of the C{%APPDATA%} environment variable
- On MacOS, it is the C{~/Library/Application Support/pyRdfa-cache}
- Otherwise, it is the C{~/.pyRdfa-cache}

This automatic choice can be overridden by the C{PyRdfaCacheDir} environment variable.

Caching can be set to be read-only, i.e., the setup might generate the cache files off-line instead of letting the tool writing its own cache when operating, e.g., as a service on the Web. This can be achieved by making the cache directory read only.

If the directories are neither readable nor writable, the vocabulary files are retrieved via HTTP every time they are hit. This may slow down processing, it is advised to avoid such a setup for the package.

The cache includes a separate index file and a file for each vocabulary file. Cache control is based upon the C{EXPIRES} header of a vocabulary file's HTTP return header: when first seen, this data is stored in the index file and controls whether the cache has to be renewed or not. If the HTTP return header does not have this entry, the date is artificially set ot the current date plus one day.

(The cache files themselves are dumped and loaded using U{Python's built in cPickle package<http://docs.python.org/release/2.7/library/pickle.html#module-cPickle>}. These are binary files. Care should be taken if they are managed by CVS: they must be declared as binary files when adding them to the repository.)

RDFa 1.1 vs. RDFa 1.0 Unfortunately, RDFa 1.1 is I{not} fully backward compatible with RDFa 1.0, meaning that, in a few cases, the triples generated from an RDFa 1.1 source are not the same as for RDFa 1.0. (See the separate U{section in the RDFa 1.1 specification<http://www.w3.org/TR/rdfa-core/#major-differences-with-rdfa-syntax-1.0>} for some further details.)

This distiller's default behavior is RDFa 1.1. However, if the source includes, in the top element of the file (e.g., the C{html} element) a C{@version} attribute whose value contains the C{RDFa 1.0} string, then the distiller switches to a RDFa 1.0 mode. (Although the C{@version} attribute is not required in RDFa 1.0, it is fairly commonly used.) Similarly, if the RDFa 1.0 DTD is used in the XHTML source, it will be taken into account (a very frequent setup is

that an XHTML file is defined with that DTD and is served as text/html; pyRdfa will consider that file as XHTML5, i.e., parse it with the HTML5 parser, but interpret the RDFa attributes under the RDFa 1.0 rules).

Transformers The package uses the concept of 'transformers': the parsed DOM tree is possibly transformed I{before} performing the real RDFa processing. This transformer structure makes it possible to add additional 'services' without distoring the core code of RDFa processing.

A transformer is a function with three arguments:

- C{node}: a DOM node for the top level element of the DOM tree
- C{options}: the current L{Options} instance
- C{state}: the current L{ExecutionContext} instance, corresponding to the top level DOM Tree element

The function may perform any type of change on the DOM tree; the typical behaviour is to add or remove attributes on specific elements. Some transformations are included in the package and can be used as examples; see the L{transform} module of the distribution. These are:

- The C{@name} attribute of the C{meta} element is copied into a C{@property} attribute of the same element
- Interpreting the 'openid' references in the header. See L{transform.OpenID} for further details.
- Implementing the Dublin Core dialect to include DC statements from the header. See L{transform.DublinCore} for further details.

The user of the package may refer add these transformers to L{Options} instance. Here is a possible usage with the "openid" tr
from pyRdfa.options import Options from pyRdfa.transform.OpenID import OpenID_transform options = Options(transformers=[OpenID_transform]) print pyRdfa(options=options).rdf_from_source('filename')

@summary: RDFa parser (distiller) @requires: Python version 2.5 or up; 2.7 is preferred @requires: U{RDFLib<http://rdflib.net>}; version 3.X is preferred. @requires: U{html5lib<http://code.google.com/p/html5lib/>} for the HTML5 parsing. @requires: U{httpheader<http://deron.meranda.us/python/httpheader/>}; however, a small modification had to make on the original file, so for this reason and to make distribution easier this module (single file) is added to the package. @organization: U{World Wide Web Consortium<http://www.w3.org>} @author: U{Ivan Herman<a href="http://www.w3.org/People/Ivan/">} @license: This software is available for use under the U{W3C® SOFTWARE NOTICE AND LICENSE<href="http://www.w3.org/Consortium/Legal/2002/copyright-software-20021231">} @copyright: W3C

@var builtInTransformers: List of built-in transformers that are to be run regardless, because they are part of the RDFa spec @var CACHE_DIR_VAR: Environment variable used to define cache directories for RDFa vocabularies in case the default setting does not work or is not appropriate. @var rdfa_current_version: Current "official" version of RDFa that this package implements by default. This can be changed at the invocation of the package @var uri_schemes: List of registered (or widely used) URI schemes; used for warnings...

exception rdflib.plugins.parsers.pyRdfa.**FailedSource**(*msg*, *http_code=None*)
Bases: *rdflib.plugins.parsers.pyRdfa.RDFaError*

Raised when the original source cannot be accessed. It does not add any new functionality to the Exception class.

__init__(*msg*, *http_code=None*)

__module__ = 'rdflib.plugins.parsers.pyRdfa'

exception rdflib.plugins.parsers.pyRdfa.**HTTPError**(*http_msg*, *http_code*)
Bases: *rdflib.plugins.parsers.pyRdfa.RDFaError*

Raised when HTTP problems are detected. It does not add any new functionality to the Exception class.

__init__(*http_msg*, *http_code*)

__module__ = 'rdflib.plugins.parsers.pyRdfa'

exception rdflib.plugins.parsers.pyRdfa.**ProcessingError**(*msg*)
Bases: *rdflib.plugins.parsers.pyRdfa.RDFaError*

Error found during processing. It does not add any new functionality to the Exception class.

__module__ = 'rdflib.plugins.parsers.pyRdfa'

exception rdflib.plugins.parsers.pyRdfa.**RDFaError**(*msg*)
Bases: exceptions.Exception

Superclass exceptions representing error conditions defined by the RDFa 1.1 specification. It does not add any new functionality to the Exception class.

__init__(*msg*)

__module__ = 'rdflib.plugins.parsers.pyRdfa'

__weakref__
list of weak references to the object (if defined)

rdflib.plugins.parsers.pyRdfa.**processURI**(*uri*, *outputFormat*, *form={}*)
The standard processing of an RDFa uri options in a form; used as an entry point from a CGI call.

The call accepts extra form options (i.e., HTTP GET options) as follows:

- C{graph=[output|processor|output,processor|processor,output]} specifying which graphs are returned. Default: C{output}
- C{space_preserve=[true|false]} means that plain literals are normalized in terms of white spaces. Default: C{false}
- C{rfa_version} provides the RDFa version that should be used for distilling. The string should be of the form "1.0" or "1.1". Default is the highest version the current package implements, currently "1.1"
- C{host_language=[xhtml,html,xml]} : the host language. Used when files are uploaded or text is added verbatim, otherwise the HTTP return header should be used. Default C{xml}
- C{embedded_rdf=[true|false]} : whether embedded turtle or RDF/XML content should be added to the output graph. Default: C{false}
- C{vocab_expansion=[true|false]} : whether the vocabularies should be expanded through the restricted RDFS entailment. Default: C{false}
- C{vocab_cache=[true|false]} : whether vocab caching should be performed or whether it should be ignored and vocabulary files should be picked up every time. Default: C{false}
- C{vocab_cache_report=[true|false]} : whether vocab caching details should be reported. Default: C{false}
- C{vocab_cache_bypass=[true|false]} : whether vocab caches have to be regenerated every time. Default: C{false}
- C{rdfa_lite=[true|false]} : whether warnings should be generated for non RDFa Lite attribute usage. Default: C{false}

@param uri: URI to access. Note that the C{text:} and C{uploaded:} fake URI values are treated separately; the former is for textual intput (in which case a StringIO is used to get the data) and the latter is for uploaded file, where the form gives access to the file directly. @param outputFormat: serialization format, as defined by the package. Currently "xml", "turtle", "nt", or "json". Default is "turtle", also used if any other string is given. @param form: extra call options (from the CGI call) to set up the local options @type form: cgi FieldStorage instance @return: serialized graph @rtype: string

class rdflib.plugins.parsers.pyRdfa.**pyRdfa**(*options=None*, *base=''*, *media_type=''*, *rdfa_version=None*)

Main processing class for the distiller

@ivar options: an instance of the L{Options} class @ivar media_type: the preferred default media type, possibly set at initialization @ivar base: the base value, possibly set at initialization @ivar http_status: HTTP Status, to be returned when the package is used via a CGI entry. Initially set to 200, may be modified by exception handlers

__init__(*options=None*, *base=''*, *media_type=''*, *rdfa_version=None*)

@keyword options: Options for the distiller @type options: L{Options} @keyword base: URI for the default "base" value (usually the URI of the file to be processed) @keyword media_type: explicit setting of the preferred media type (a.k.a. content type) of the the RDFa source @keyword rdfa_version: the RDFa version that should be used. If not set, the value of the global L{rdfa_current_version} variable is used

__module__ = 'rdflib.plugins.parsers.pyRdfa'

graph_from_DOM(*dom*, *graph=None*, *pgraph=None*)

Extract the RDF Graph from a DOM tree. This is where the real processing happens. All other methods get down to this one, eventually (e.g., after opening a URI and parsing it into a DOM). @param dom: a DOM Node element, the top level entry node for the whole tree (i.e., the C{dom.documentElement} is used to initiate processing down the node hierarchy) @keyword graph: an RDF Graph (if None, than a new one is created) @type graph: rdflib Graph instance. @keyword pgraph: an RDF Graph to hold (possibly) the processor graph content. If None, and the error/warning triples are to be generated, they will be added to the returned graph. Otherwise they are stored in this graph. @type pgraph: rdflib Graph instance @return: an RDF Graph @rtype: rdflib Graph instance

graph_from_source(*name*, *graph=None*, *rdfOutput=False*, *pgraph=None*)

Extract an RDF graph from an RDFa source. The source is parsed, the RDF extracted, and the RDFa Graph is returned. This is a front-end to the L{pyRdfa.graph_from_DOM} method.

@param name: a URI, a file name, or a file-like object @param graph: rdflib Graph instance. If None, a new one is created. @param pgraph: rdflib Graph instance for the processor graph. If None, and the error/warning triples are to be generated, they will be added to the returned graph. Otherwise they are stored in this graph. @param rdfOutput: whether runtime exceptions should be turned into RDF and returned as part of the processor graph @return: an RDF Graph @rtype: rdflib Graph instance

rdf_from_source(*name*, *outputFormat='turtle'*, *rdfOutput=False*)

Extract and RDF graph from an RDFa source and serialize it in one graph. The source is parsed, the RDF extracted, and serialization is done in the specified format. @param name: a URI, a file name, or a file-like object @keyword outputFormat: serialization format. Can be one of "turtle", "n3", "xml", "pretty-xml", "nt". "xml", "pretty-xml", "json" or "json-ld". "turtle" and "n3", "xml" and "pretty-xml", and "json" and "json-ld" are synonyms, respectively. Note that the JSON-LD serialization works with RDFLib 3.* only. @keyword rdfOutput: controls what happens in case an exception is raised. If the value is False, the caller is responsible handling it; otherwise a graph is returned with an error message included in the processor graph @type rdfOutput: boolean @return: a serialized RDF Graph @rtype: string

rdf_from_sources(*names*, *outputFormat='turtle'*, *rdfOutput=False*)

Extract and RDF graph from a list of RDFa sources and serialize them in one graph. The sources are parsed, the RDF extracted, and serialization is done in the specified format. @param names: list of sources, each can be a URI, a file name, or a file-like object @keyword outputFormat: serialization format. Can be one of "turtle", "n3", "xml", "pretty-xml", "nt". "xml", "pretty-xml", "json" or "json-ld". "turtle" and "n3", "xml" and "pretty-xml", and "json" and "json-ld" are synonyms, respectively. Note that the JSON-LD serialization works with RDFLib 3.* only. @keyword rdfOutput: controls what happens in case an exception is raised. If the value is False, the caller is responsible handling it; otherwise a graph is returned with an error message included in the processor graph @type rdfOutput: boolean @return: a serialized RDF Graph @rtype: string

exception rdflib.plugins.parsers.pyRdfa.**pyRdfaError**
Bases: exceptions.Exception

Superclass exceptions representing error conditions outside the RDFa 1.1 specification.

__module__ = 'rdflib.plugins.parsers.pyRdfa'

__weakref__
list of weak references to the object (if defined)

embeddedRDF Module Extracting possible embedded RDF/XML content from the file and parse it separately into the Graph. This is used, for example by U{SVG 1.2 Tiny<http://www.w3.org/TR/SVGMobile12/>}.

@author: U{Ivan Herman<a href="http://www.w3.org/People/Ivan/">} @license: This software is available for use under the U{W3C® SOFTWARE NOTICE AND LICENSE<href="http://www.w3.org/Consortium/Legal/2002/copyright-software-20021231">} @contact: Ivan Herman, ivan@w3.org @version: $Id: embeddedRDF.py,v 1.15 2012/11/16 17:51:53 ivan Exp $

rdflib.plugins.parsers.pyRdfa.embeddedRDF.**handle_embeddedRDF**(*node*, *graph*, *state*)
Handles embedded RDF. There are two possibilities:

- the file is one of the XML dialects that allows for an embedded RDF/XML portion. See the L{host.accept_embedded_rdf_xml} for those (a typical example is SVG).
- the file is HTML and there is a turtle portion in the C{<script>} element with type text/turtle.

@param node: a DOM node for the top level element @param graph: target rdf graph @type graph: RDFLib's Graph object instance @param state: the inherited state (namespaces, lang, etc) @type state: L{state.ExecutionContext} @return: whether an RDF/XML or turtle content has been detected or not. If TRUE, the RDFa processing should not occur on the node and its descendents. @rtype: Boolean

initialcontext Module Built-in version of the initial contexts for RDFa Core, and RDFa + HTML

@summary: Management of vocabularies, terms, and their mapping to URI-s. @requires: U{RDFLib package<http://rdflib.net>} @organization: U{World Wide Web Consortium<http://www.w3.org>} @author: U{Ivan Herman<a href="http://www.w3.org/People/Ivan/">} @license: This software is available for use under the U{W3C® SOFTWARE NOTICE AND LICENSE<href="http://www.w3.org/Consortium/Legal/2002/copyright-software-20021231">}

@var initial_context: dictionary for all the initial context data, keyed through the context URI-s

class rdflib.plugins.parsers.pyRdfa.initialcontext.**Wrapper**

__module__ = 'rdflib.plugins.parsers.pyRdfa.initialcontext'

options Module L{Options} class: collect the possible options that govern the parsing possibilities. The module also includes the L{ProcessorGraph} class that handles the processor graph, per RDFa 1.1 (i.e., the graph containing errors and warnings).

@summary: RDFa parser (distiller) @requires: U{RDFLib<http://rdflib.net>} @organization: U{World Wide Web Consortium<http://www.w3.org>} @author: U{Ivan Herman<a href="http://www.w3.org/People/Ivan/">} @license: This software is available for use under the U{W3C SOFTWARE NOTICE AND LICENSE<href="http://www.w3.org/Consortium/Legal/2002/copyright-software-20021231">}

class rdflib.plugins.parsers.pyRdfa.options.**Options**(*output_default_graph=True*, *output_processor_graph=False*, *space_preserve=True*, *transformers=[]*, *embedded_rdf=True*, *vocab_expansion=False*, *vocab_cache=True*, *vocab_cache_report=False*, *refresh_vocab_cache=False*, *add_informational_messages=False*, *check_lite=False*, *experimental_features=False*)

Settable options. An instance of this class is stored in the L{execution context<ExecutionContext>} of the parser.

@ivar space_preserve: whether plain literals should preserve spaces at output or not @type space_preserve: Boolean

@ivar output_default_graph: whether the 'default' graph should be returned to the user @type output_default_graph: Boolean

@ivar output_processor_graph: whether the 'processor' graph should be returned to the user @type output_processor_graph: Boolean

@ivar processor_graph: the 'processor' Graph @type processor_graph: L{ProcessorGraph}

@ivar transformers: extra transformers @type transformers: list

@ivar vocab_cache_report: whether the details of vocabulary file caching process should be reported as information (mainly for debug) @type vocab_cache_report: Boolean

@ivar refresh_vocab_cache: whether the caching checks of vocabs should be by-passed, ie, if caches should be re-generated regardless of the stored date (important for vocab development) @type refresh_vocab_cache: Boolean

@ivar embedded_rdf: whether embedded RDF (ie, turtle in an HTML script element or an RDF/XML content in SVG) should be extracted and added to the final graph. This is a non-standard option... @type embedded_rdf: Boolean

@ivar vocab_expansion: whether the @vocab elements should be expanded and a mini-RDFS processing should be done on the merged graph @type vocab_expansion: Boolean

@ivar vocab_cache: whether the system should use the vocabulary caching mechanism when expanding via the mini-RDFS, or should just fetch the graphs every time @type vocab_cache: Boolean

@ivar host_language: the host language for the RDFa attributes. Default is HostLanguage.xhtml, but it can be HostLanguage.rdfa_core and HostLanguage.html5, or others... @type host_language: integer (logically: an enumeration)

@ivar content_type: the content type of the host file. Default is None @type content_type: string (logically: an enumeration)

@ivar add_informational_messages: whether informational messages should also be added to the processor graph, or only errors and warnings

@ivar experimental_features: whether experimental features should be activated; that is a developer's option... @ivar check_lite: whether RDFa Lite should be checked, to generate warnings.

__init__(*output_default_graph=True*, *output_processor_graph=False*, *space_preserve=True*, *transformers=[]*, *embedded_rdf=True*, *vocab_expansion=False*, *vocab_cache=True*, *vocab_cache_report=False*, *refresh_vocab_cache=False*, *add_informational_messages=False*, *check_lite=False*, *experimental_features=False*)

__module__ = 'rdflib.plugins.parsers.pyRdfa.options'

__str__()

add_error(*txt*, *err_type=None*, *context=None*, *node=None*, *buggy_value=None*)
Add an error to the processor graph. @param txt: the information text. @keyword err_type: Error Class @type err_type: URIRef @keyword context: possible context to be added to the processor graph @type context: URIRef or String @keyword buggy_value: a special case when a 'term' is not recognized; no error is generated for that case if the value is part of the 'usual' XHTML terms, because almost all RDFa file contains some of those and that would pollute the output @type buggy_value: String

add_info(*txt*, *info_type=None*, *context=None*, *node=None*, *buggy_value=None*)
Add an informational comment to the processor graph. @param txt: the information text. @keyword info_type: Info Class @type info_type: URIRef @keyword context: possible context to be added to the processor graph @type context: URIRef or String @keyword buggy_value: a special case when a 'term' is not recognized; no information is generated for that case if the value is part of the 'usual' XHTML terms, because almost all RDFa file contains some of those and that would pollute the output @type buggy_value: String

add_warning(*txt*, *warning_type=None*, *context=None*, *node=None*, *buggy_value=None*)
Add a warning to the processor graph. @param txt: the warning text. @keyword warning_type: Warning Class @type warning_type: URIRef @keyword context: possible context to be added to the processor graph @type context: URIRef or String @keyword buggy_value: a special case when a 'term' is not recognized; no warning is generated for that case if the value is part of the 'usual' XHTML terms, because almost all RDFa file contains some of those and that would pollute the output @type buggy_value: String

reset_processor_graph()
Empty the processor graph. This is necessary if the same options is reused for several RDFa sources, and new error messages should be generated.

set_host_language(*content_type*)
Set the host language for processing, based on the recognized types. If this is not a recognized content type, it falls back to RDFa core (i.e., XML) @param content_type: content type @type content_type: string

class rdflib.plugins.parsers.pyRdfa.options.**ProcessorGraph**
Wrapper around the 'processor graph', ie, the (RDF) Graph containing the warnings, error messages, and informational messages.

__init__()

__module__ = 'rdflib.plugins.parsers.pyRdfa.options'

add_http_context(*subj*, *http_code*)
Add an additional HTTP context to a message with subject in C{subj}, using the U{<http://www.w3.org/2006/http#>} vocabulary. Typically used to extend an error structure, as created by L{add_triples}.

@param subj: an RDFLib resource, typically a blank node @param http_code: HTTP status code

add_triples(*msg*, *top_class*, *info_class*, *context*, *node*)
Add an error structure to the processor graph: a bnode with a number of predicates. The structure follows U{the processor graph vocabulary<http://www.w3.org/2010/02/rdfa/wiki/Processor_Graph_Vocabulary>} as described on the RDFa WG Wiki page.

@param msg: the core error message, added as an object to a dc:description @param top_class: Error, Warning, or Info; an explicit rdf:type added to the bnode @type top_class: URIRef @param info_class: An additional error class, added as an rdf:type to the bnode in case it is not None @type info_class: URIRef @param context: An additional information added, if not None, as an object with rdfa:context as

a predicate @type context: either an URIRef or a URI String (an URIRef will be created in the second case) @param node: The node's element name that contains the error @type node: string @return: the bnode that serves as a subject for the errors. The caller may add additional information @rtype: BNode

parse Module The core parsing function of RDFa. Some details are put into other modules to make it clearer to update/modify (e.g., generation of C{@property} values, or managing the current state).

Note that the entry point (L{parse_one_node}) bifurcates into an RDFa 1.0 and RDFa 1.1 version, ie, to L{_parse_1_0} and L{_parse_1_1}. Some of the parsing details (management of C{@property}, list facilities, changed behavior on C{@typeof})) have changed between versions and forcing the two into one function would be counter productive.

@summary: RDFa core parser processing step @organization: U{World Wide Web Consortium<http://www.w3.org>} @author: U{Ivan Herman<a href="http://www.w3.org/People/Ivan/">} @license: This software is available for use under the U{W3C® SOFTWARE NOTICE AND LICENSE<href="http://www.w3.org/Consortium/Legal/2002/copyright-software-20021231">}

rdflib.plugins.parsers.pyRdfa.parse.**handle_role_attribute**(*node*, *graph*, *state*)

Handling the role attribute, according to http://www.w3.org/TR/role-attribute/#using-role-in-conjunction-with-rdfa @param node: the DOM node to handle @param graph: the RDF graph @type graph: RDFLib's Graph object instance @param state: the inherited state (namespaces, lang, etc.) @type state: L{state.ExecutionContext}

rdflib.plugins.parsers.pyRdfa.parse.**parse_one_node**(*node*, *graph*, *parent_object*, *incoming_state*, *parent_incomplete_triples*)

The (recursive) step of handling a single node.

This entry just switches between the RDFa 1.0 and RDFa 1.1 versions for parsing. This method is only invoked once, actually, from the top level; the recursion then happens in the L{_parse_1_0} and L{_parse_1_1} methods for RDFa 1.0 and RDFa 1.1, respectively.

@param node: the DOM node to handle @param graph: the RDF graph @type graph: RDFLib's Graph object instance @param parent_object: the parent's object, as an RDFLib URIRef @param incoming_state: the inherited state (namespaces, lang, etc.) @type incoming_state: L{state.ExecutionContext} @param parent_incomplete_triples: list of hanging triples (the missing resource set to None) to be handled (or not) by the current node. @return: whether the caller has to complete it's parent's incomplete triples @rtype: Boolean

property Module Implementation of the C{@property} value handling.

RDFa 1.0 and RDFa 1.1 are fairly different. RDFa 1.0 generates only literals, see U{RDFa Task Force's wiki page<http://www.w3.org/2006/07/SWD/wiki/RDFa/LiteralObject>} for the details. On the other hand, RDFa 1.1, beyond literals, can also generate URI references. Hence the duplicate method in the L{ProcessProperty} class, one for RDFa 1.0 and the other for RDFa 1.1.

@summary: RDFa Literal generation @requires: U{RDFLib package<http://rdflib.net>} @organization: U{World Wide Web Consortium<http://www.w3.org>} @author: U{Ivan Herman<a href="http://www.w3.org/People/Ivan/">} @license: This software is available for use under the U{W3C® SOFTWARE NOTICE AND LICENSE<href="http://www.w3.org/Consortium/Legal/2002/copyright-software-20021231">}

class rdflib.plugins.parsers.pyRdfa.property.**ProcessProperty**(*node*, *graph*, *subject*, *state*, *typed_resource=None*)

Generate the value for C{@property} taking into account datatype, etc. Note: this class is created only if the C{@property} is indeed present, no need to check.

@ivar node: DOM element node @ivar graph: the (RDF) graph to add the properies to @ivar subject: the RDFLib URIRef serving as a subject for the generated triples @ivar state: the current state to be used for the

CURIE-s @type state: L{state.ExecutionContext} @ivar typed_resource: Typically the bnode generated by a @typeof

__init__(*node*, *graph*, *subject*, *state*, *typed_resource=None*)
@param node: DOM element node @param graph: the (RDF) graph to add the properies to @param subject: the RDFLib URIRef serving as a subject for the generated triples @param state: the current state to be used for the CURIE-s @param state: L{state.ExecutionContext} @param typed_resource: Typically the bnode generated by a @typeof; in RDFa 1.1, that becomes the object for C{@property}

__module__ = 'rdflib.plugins.parsers.pyRdfa.property'

generate()
Common entry point for the RDFa 1.0 and RDFa 1.1 versions; bifurcates based on the RDFa version, as retrieved from the state object.

generate_1_0()
Generate the property object, 1.0 version

generate_1_1()
Generate the property object, 1.1 version

state Module Parser's execution context (a.k.a. state) object and handling. The state includes:

- language, retrieved from C{@xml:lang} or C{@lang}
- URI base, determined by C{<base>} or set explicitly. This is a little bit superfluous, because the current RDFa syntax does not make use of C{@xml:base}; i.e., this could be a global value. But the structure is prepared to add C{@xml:base} easily, if needed.
- options, in the form of an L{options<pyRdfa.options>} instance
- a separate vocabulary/CURIE handling resource, in the form of an L{termorcurie<pyRdfa.TermOrCurie>} instance

The execution context object is also used to handle URI-s, CURIE-s, terms, etc.

@summary: RDFa parser execution context @organization: U{World Wide Web Consortium<http://www.w3.org>} @author: U{Ivan Herman<a href="http://www.w3.org/People/Ivan/">} @license: This software is available for use under the U{W3C® SOFTWARE NOTICE AND LICENSE<href="http://www.w3.org/Consortium/Legal/2002/copyright-software-20021231">}

class rdflib.plugins.parsers.pyRdfa.state.**ExecutionContext**(*node*, *graph*, *inherited_state=None*, *base=''*, *options=None*, *rdfa_version=None*)
State at a specific node, including the current set of namespaces in the RDFLib sense, current language, the base, vocabularies, etc. The class is also used to interpret URI-s and CURIE-s to produce URI references for RDFLib.

@ivar options: reference to the overall options @type options: L{Options} @ivar base: the 'base' URI @ivar parsedBase: the parsed version of base, as produced by urlparse.urlsplit @ivar defaultNS: default namespace (if defined via @xmlns) to be used for XML Literals @ivar lang: language tag (possibly None) @ivar term_or_curie: vocabulary management class instance @type term_or_curie: L{termorcurie.TermOrCurie} @ivar list_mapping: dictionary of arrays, containing a list of URIs key-ed via properties for lists @ivar node: the node to which this state belongs @type node: DOM node instance @ivar rdfa_version: RDFa version of the content @type rdfa_version: String @ivar supress_lang: in some cases, the effect of the lang attribute should be supressed for the given node, although it should be inherited down below (example: @value attribute of the data element in HTML5) @type supress_lang: Boolean @cvar _list: list of attributes that allow for lists of values and should be treated as such @cvar _resource_type: dictionary; mapping table from attribute name to the exact method to retrieve the URI(s). Is initialized at first instantiation.

__init__(*node*, *graph*, *inherited_state=None*, *base=''*, *options=None*, *rdfa_version=None*)
@param node: the current DOM Node @param graph: the RDFLib Graph @keyword inherited_state: the state as inherited from upper layers. This inherited_state is mixed with the state information retrieved from the current node. @type inherited_state: L{state.ExecutionContext} @keyword base: string denoting the base URI for the specific node. This overrides the possible base inherited from the upper layers. The current XHTML+RDFa syntax does not allow the usage of C{@xml:base}, but SVG1.2 does, so this is necessary for SVG (and other possible XML dialects that accept C{@xml:base}) @keyword options: invocation options, and references to warning graphs @type options: L{Options<pyRdfa.options>}

__module__ = 'rdflib.plugins.parsers.pyRdfa.state'

add_to_list_mapping(*property*, *resource*)
Add a new property-resource on the list mapping structure. The latter is a dictionary of arrays; if the array does not exist yet, it will be created on the fly.

@param property: the property URI, used as a key in the dictionary @param resource: the resource to be added to the relevant array in the dictionary. Can be None; this is a dummy placeholder for C{<span rel="property" inlist>...} constructions that may be filled in by children or siblings; if not an empty list has to be generated.

getResource(**args*)
Get single resources from several different attributes. The first one that returns a valid URI wins. @param args: variable list of attribute names, or a single attribute being a list itself. @return: an RDFLib URIRef instance (or None) :

getURI(*attr*)
Get the URI(s) for the attribute. The name of the attribute determines whether the value should be a pure URI, a CURIE, etc, and whether the return is a single element of a list of those. This is done using the L{ExecutionContext._resource_type} table. @param attr: attribute name @type attr: string @return: an RDFLib URIRef instance (or None) or a list of those

get_list_origin()
Return the origin of the list, ie, the subject to attach the final list(s) to @return: URIRef

get_list_props()
Return the list of property values in the list structure @return: list of URIRef

get_list_value(*prop*)
Return the list of values in the list structure for a specific property @return: list of RDF nodes

list_empty()
Checks whether the list is empty. @return: Boolean

reset_list_mapping(*origin=None*)
Reset, ie, create a new empty dictionary for the list mapping.

set_list_origin(*origin*)
Set the origin of the list, ie, the subject to attach the final list(s) to @param origin: URIRef

class rdflib.plugins.parsers.pyRdfa.state.**ListStructure**
Special class to handle the C{@inlist} type structures in RDFa 1.1; stores the "origin", i.e, where the list will be attached to, and the mappings as defined in the spec.

__init__()

__module__ = 'rdflib.plugins.parsers.pyRdfa.state'

termorcurie Module Management of vocabularies, terms, and their mapping to URI-s. The main class of this module (L{TermOrCurie}) is, conceptually, part of the overall state of processing at a node (L{state.ExecutionContext}) but putting it into a separate module makes it easider to maintain.

@summary: Management of vocabularies, terms, and their mapping to URI-s. @requires: U{RDFLib package<http://rdflib.net>} @organization: U{World Wide Web Consortium<http://www.w3.org>} @author: U{Ivan Herman<a href="http://www.w3.org/People/Ivan/">} @license: This software is available for use under the U{W3C® SOFTWARE NOTICE AND LICENSE<href="http://www.w3.org/Consortium/Legal/2002/copyright-software-20021231">}

@var XHTML_PREFIX: prefix for the XHTML vocabulary URI (set to 'xhv') @var XHTML_URI: URI prefix of the XHTML vocabulary @var ncname: Regular expression object for NCNAME @var termname: Regular expression object for a term @var xml_application_media_type: Regular expression object for a general XML application media type

class rdflib.plugins.parsers.pyRdfa.termorcurie.**InitialContext**(*state*, *top_level*)

Get the initial context values. In most cases this class has an empty content, except for the top level (in case of RDFa 1.1). Each L{TermOrCurie} class has one instance of this class. It provides initial mappings for terms, namespace prefixes, etc, that the top level L{TermOrCurie} instance uses for its own initialization.

@ivar terms: collection of all term mappings @type terms: dictionary @ivar ns: namespace mapping @type ns: dictionary @ivar vocabulary: default vocabulary @type vocabulary: string

__init__(*state*, *top_level*)

@param state: the state behind this term mapping @type state: L{state.ExecutionContext} @param top_level : whether this is the top node of the DOM tree (the only place where initial contexts are handled) @type top_level : boolean

__module__ = 'rdflib.plugins.parsers.pyRdfa.termorcurie'

class rdflib.plugins.parsers.pyRdfa.termorcurie.**TermOrCurie**(*state*, *graph*, *inherited_state*)

Wrapper around vocabulary management, ie, mapping a term to a URI, as well as a CURIE to a URI. Each instance of this class belongs to a "state", instance of L{state.ExecutionContext}. Context definitions are managed at initialization time.

(In fact, this class is, conceptually, part of the overall state at a node, and has been separated here for an easier maintenance.)

The class takes care of the stack-like behavior of vocabulary items, ie, inheriting everything that is possible from the "parent". At initialization time, this works through the prefix definitions (i.e., C{@prefix} or C{@xmln:} attributes) and/or C{@vocab} attributes.

@ivar state: State to which this instance belongs @type state: L{state.ExecutionContext} @ivar graph: The RDF Graph under generation @type graph: rdflib.Graph @ivar terms: mapping from terms to URI-s @type terms: dictionary @ivar ns: namespace declarations, ie, mapping from prefixes to URIs @type ns: dictionary @ivar default_curie_uri: URI for a default CURIE

CURIE_to_URI(*val*)

CURIE to URI mapping.

This method does I{not} take care of the last step of CURIE processing, ie, the fact that if it does not have a CURIE then the value is used a URI. This is done on the caller's side, because this has to be combined with base, for example. The method I{does} take care of BNode processing, though, ie, CURIE-s of the form "_:XXX".

@param val: the full CURIE @type val: string @return: URIRef of a URI or None.

__init__(*state*, *graph*, *inherited_state*)

Initialize the vocab bound to a specific state. @param state: the state to which this vocab instance belongs to @type state: L{state.ExecutionContext} @param graph: the RDF graph being worked on @type graph: rdflib.Graph @param inherited_state: the state inherited by the current state. 'None' if this is the top level state. @type inherited_state: L{state.ExecutionContext}

__module__ = 'rdflib.plugins.parsers.pyRdfa.termorcurie'

term_to_URI(*term*)
A term to URI mapping, where term is a simple string and the corresponding URI is defined via the @vocab (ie, default term uri) mechanism. Returns None if term is not defined @param term: string @return: an RDFLib URIRef instance (or None)

utils Module Various utilities for pyRdfa.

Most of the utilities are straightforward.

@organization: U{World Wide Web Consortium<http://www.w3.org>} @author: U{Ivan Herman<a href="http://www.w3.org/People/Ivan/">} @license: This software is available for use under the U{W3C® SOFTWARE NOTICE AND LICENSE<href="http://www.w3.org/Consortium/Legal/2002/copyright-software-20021231">}

class rdflib.plugins.parsers.pyRdfa.utils.**URIOpener**(*name*, *additional_headers={}*)
A wrapper around the urllib2 method to open a resource. Beyond accessing the data itself, the class sets a number of instance variable that might be relevant for processing. The class also adds an accept header to the outgoing request, namely text/html and application/xhtml+xml (unless set explicitly by the caller).

If the content type is set by the server, the relevant HTTP response field is used. Otherwise, common suffixes are used (see L{host.preferred_suffixes}) to set the content type (this is really of importance for C{file:///} URI-s). If none of these works, the content type is empty.

Interpretation of the content type for the return is done by Deron Meranda's U{httpheader module<http://deron.meranda.us/>}.

@ivar data: the real data, ie, a file-like object @ivar headers: the return headers as sent back by the server @ivar content_type: the content type of the resource or the empty string, if the content type cannot be determined @ivar location: the real location of the data (ie, after possible redirection and content negotiation) @ivar last_modified_date: sets the last modified date if set in the header, None otherwise @ivar expiration_date: sets the expiration date if set in the header, I{current UTC plus one day} otherwise (this is used for caching purposes, hence this artificial setting)

CONTENT_LOCATION = 'Content-Location'

CONTENT_TYPE = 'Content-Type'

EXPIRES = 'Expires'

LAST_MODIFIED = 'Last-Modified'

__init__(*name*, *additional_headers={}*)
@param name: URL to be opened @keyword additional_headers: additional HTTP request headers to be added to the call

__module__ = 'rdflib.plugins.parsers.pyRdfa.utils'

rdflib.plugins.parsers.pyRdfa.utils.**create_file_name**(*uri*)
Create a suitable file name from an (absolute) URI. Used, eg, for the generation of a file name for a cached vocabulary file.

rdflib.plugins.parsers.pyRdfa.utils.**dump**(*node*)
This is just for debug purposes: it prints the essential content of the node in the tree starting at node.

@param node: DOM node

rdflib.plugins.parsers.pyRdfa.utils.**has_one_of_attributes**(*node*, **args*)
Check whether one of the listed attributes is present on a (DOM) node. @param node: DOM element node @param args: possible attribute names @return: True or False @rtype: Boolean

rdflib.plugins.parsers.pyRdfa.utils.**quote_URI**(*uri*, *options=None*)
'quote' a URI, ie, exchange special characters for their '%..' equivalents. Some of the characters may stay as they are (listed in L{_unquotedChars}. If one of the characters listed in L{_warnChars} is also in the uri, an extra warning is also generated. @param uri: URI @param options: @type options: L{Options<pyRdfa.Options>}

rdflib.plugins.parsers.pyRdfa.utils.**return_XML**(*state*, *inode*, *base=True*, *xmlns=True*)
Get (recursively) the XML Literal content of a DOM Element Node. (Most of the processing is done via a C{node.toxml} call of the xml minidom implementation.)

@param inode: DOM Node @param state: L{pyRdfa.state.ExecutionContext} @param base: whether the base element should be added to the output @type base: Boolean @param xmlns: whether the namespace declarations should be repeated in the generated node @type xmlns: Boolean @return: string

rdflib.plugins.parsers.pyRdfa.utils.**traverse_tree**(*node*, *func*)
Traverse the whole element tree, and perform the function C{func} on all the elements. @param node: DOM element node @param func: function to be called on the node. Input parameter is a DOM Element Node. If the function returns a boolean True, the recursion is stopped.

Subpackages

extras Package

extras Package Collection of external modules that are used by pyRdfa and are added for an easier distribution

httpheader Module Utility functions to work with HTTP headers.

This module provides some utility functions useful for parsing and dealing with some of the HTTP 1.1 protocol headers which are not adequately covered by the standard Python libraries.

Requires Python 2.2 or later.

The functionality includes the correct interpretation of the various Accept-* style headers, content negotiation, byte range requests, HTTP-style date/times, and more.

There are a few classes defined by this module:

- class content_type – media types such as 'text/plain'
- class language_tag – language tags such as 'en-US'
- class range_set – a collection of (byte) range specifiers
- class range_spec – a single (byte) range specifier

The primary functions in this module may be categorized as follows:

- Content negotiation functions... * acceptable_content_type() * acceptable_language() * acceptable_charset() * acceptable_encoding()
- Mid-level header parsing functions... * parse_accept_header() * parse_accept_language_header() * parse_range_header()
- Date and time... * http_datetime() * parse_http_datetime()
- Utility functions... * quote_string() * remove_comments() * canonical_charset()
- Low level string parsing functions... * parse_comma_list() * parse_comment() * parse_qvalue_accept_list() * parse_media_type() * parse_number() * parse_parameter_list() * parse_quoted_string() * parse_range_set() * parse_range_spec() * parse_token() * parse_token_or_quoted_string()

And there are some specialized exception classes:

- RangeUnsatisfiableError
- RangeUnmergableError
- ParseError

See also:

- **RFC 2616, "Hypertext Transfer Protocol – HTTP/1.1", June 1999.**
  <http://www.ietf.org/rfc/rfc2616.txt> Errata at <http://purl.org/NET/http-errata>
- **RFC 2046, "(MIME) Part Two: Media Types", November 1996.**
  <http://www.ietf.org/rfc/rfc2046.txt>
- **RFC 3066, "Tags for the Identification of Languages", January 2001.**
  <http://www.ietf.org/rfc/rfc3066.txt>

Note: I have made a small modification on the regexp for internet date, to make it more liberal (ie, accept a time zone string of the form +0000) Ivan Herman <http://www.ivan-herman.net>, March 2011.

Have added statements to make it (hopefully) Python 3 compatible. Ivan Herman <http://www.ivan-herman.net>, August 2012.

exception rdflib.plugins.parsers.pyRdfa.extras.httpheader.**ParseError**(*args*, *input_string*, *at_position*)

Bases: exceptions.ValueError

Exception class representing a string parsing error.

__init__(*args*, *input_string*, *at_position*)

__module__ = 'rdflib.plugins.parsers.pyRdfa.extras.httpheader'

__str__()

exception rdflib.plugins.parsers.pyRdfa.extras.httpheader.**RangeUnmergableError**(*reason=None*)

Bases: exceptions.ValueError

Exception class when byte ranges are noncontiguous and can not be merged together.

__init__(*reason=None*)

__module__ = 'rdflib.plugins.parsers.pyRdfa.extras.httpheader'

exception rdflib.plugins.parsers.pyRdfa.extras.httpheader.**RangeUnsatisfiableError**(*reason=None*)

Bases: exceptions.ValueError

Exception class when a byte range lies outside the file size boundaries.

__init__(*reason=None*)

__module__ = 'rdflib.plugins.parsers.pyRdfa.extras.httpheader'

rdflib.plugins.parsers.pyRdfa.extras.httpheader.**acceptable_charset**(*accept_charset_header*, *charsets*, *ignore_wildcard=True*, *default='ISO-8859-1'*)

Determines if the given charset is acceptable to the user agent.

The accept_charset_header should be the value present in the HTTP "Accept-Charset:" header. In mod_python this is typically obtained from the req.http_headers table; in WSGI it is environ["Accept-Charset"]; other web frameworks may provide other methods of obtaining it.

Optionally the accept_charset_header parameter can instead be the list returned from the parse_accept_header() function in this module.

The charsets argument should either be a charset identifier string, or a sequence of them.

This function returns the charset identifier string which is the most prefered and is acceptable to both the user agent and the caller. It will return the default value if no charset is negotiable.

Note that the wildcarded charset "*" will be ignored. To override this, call with ignore_wildcard=False.

See also: RFC 2616 section 14.2, and <http://www.iana.org/assignments/character-sets>

rdflib.plugins.parsers.pyRdfa.extras.httpheader.**acceptable_content_type**(*accept_header*, *content_types*, *ignore_wildcard=True*)

Determines if the given content type is acceptable to the user agent.

The accept_header should be the value present in the HTTP "Accept:" header. In mod_python this is typically obtained from the req.http_headers_in table; in WSGI it is environ["Accept"]; other web frameworks may provide other methods of obtaining it.

Optionally the accept_header parameter can be pre-parsed, as returned from the parse_accept_header() function in this module.

The content_types argument should either be a single MIME media type string, or a sequence of them. It represents the set of content types that the caller (server) is willing to send. Generally, the server content_types should not contain any wildcarded values.

This function determines which content type which is the most preferred and is acceptable to both the user agent and the server. If one is negotiated it will return a four-valued tuple like:

(server_content_type, ua_content_range, qvalue, accept_parms)

The first tuple value is one of the server's content_types, while the remaining tuple values descript which of the client's acceptable content_types was matched. In most cases accept_parms will be an empty list (see description of parse_accept_header() for more details).

If no content type could be negotiated, then this function will return None (and the caller should typically cause an HTTP 406 Not Acceptable as a response).

Note that the wildcarded content type "/" sent by the client will be ignored, since it is often incorrectly sent by web browsers that don't really mean it. To override this, call with ignore_wildcard=False. Partial wildcards such as "image/*" will always be processed, but be at a lower priority than a complete matching type.

See also: RFC 2616 section 14.1, and <http://www.iana.org/assignments/media-types/>

rdflib.plugins.parsers.pyRdfa.extras.httpheader.**acceptable_language**(*accept_header*, *server_languages*, *ignore_wildcard=True*, *assume_superiors=True*)

Determines if the given language is acceptable to the user agent.

The accept_header should be the value present in the HTTP "Accept-Language:" header. In mod_python this is typically obtained from the req.http_headers_in table; in WSGI it is environ["Accept-Language"]; other web frameworks may provide other methods of obtaining it.

Optionally the accept_header parameter can be pre-parsed, as returned by the parse_accept_language_header() function defined in this module.

The server_languages argument should either be a single language string, a language_tag object, or a sequence of them. It represents the set of languages that the server is willing to send to the user agent.

Note that the wildcarded language tag "*" will be ignored. To override this, call with ignore_wildcard=False, and even then it will be the lowest-priority choice regardless of it's quality factor (as per HTTP spec).

If the assume_superiors is True then it the languages that the browser accepts will automatically include all superior languages. Any superior languages which must be added are done so with one half the qvalue of the language which is present. For example, if the accept string is "en-US", then it will be treated as if it were "en-US, en;q=0.5". Note that although the HTTP 1.1 spec says that browsers are supposed to encourage users to configure all acceptable languages, sometimes they don't, thus the ability for this function to assume this. But setting assume_superiors to False will insure strict adherence to the HTTP 1.1 spec; which means that if the browser accepts "en-US", then it will not be acceptable to send just "en" to it.

This function returns the language which is the most prefered and is acceptable to both the user agent and the caller. It will return None if no language is negotiable, otherwise the return value is always an instance of language_tag.

See also: RFC 3066 <http://www.ietf.org/rfc/rfc3066.txt>, and ISO 639, links at <http://en.wikipedia.org/wiki/ISO_639>, and <http://www.iana.org/assignments/language-tags>.

rdflib.plugins.parsers.pyRdfa.extras.httpheader.**canonical_charset**(*charset*)

Returns the canonical or preferred name of a charset.

Additional character sets can be recognized by this function by altering the character_set_aliases dictionary in this module. Charsets which are not recognized are simply converted to upper-case (as charset names are always case-insensitive).

See <http://www.iana.org/assignments/character-sets>.

class rdflib.plugins.parsers.pyRdfa.extras.httpheader.**content_type**(*content_type_string=None*, *with_parameters=True*)

Bases: object

This class represents a media type (aka a MIME content type), including parameters.

You initialize these by passing in a content-type declaration string, such as "text/plain; charset=ascii", to the constructor or to the set() method. If you provide no string value, the object returned will represent the wildcard */* content type.

Normally you will get the value back by using str(), or optionally you can access the components via the 'major', 'minor', 'media_type', or 'parmdict' members.

__eq__(*other*)

Equality test.

Note that this is an exact match, including any parameters if any.

__getstate__()

Pickler

__hash__()

Hash this object; the hash is dependent only upon the value.

__init__(*content_type_string=None*, *with_parameters=True*)

Create a new content_type object.

See the set() method for a description of the arguments.

__len__()

Logical length of this media type. For example:

len('/') -> 0 len('image/*') -> 1 len('image/png') -> 2 len('text/plain; charset=utf-8') -> 3 len('text/plain; charset=utf-8; filename=xyz.txt') -> 4

__module__ = 'rdflib.plugins.parsers.pyRdfa.extras.httpheader'

__ne__(*other*)
Inequality test.

__repr__()
Python representation of this object.

__setstate__(*state*)
Unpickler

__str__()
String value.

__unicode__()
Unicode string value.

is_composite()
Is this media type composed of multiple parts.

is_universal_wildcard()
Returns True if this is the unspecified '/' media type.

is_wildcard()
Returns True if this is a 'something/*' media type.

is_xml()
Returns True if this media type is XML-based.

Note this does not consider text/html to be XML, but application/xhtml+xml is.

major
Major media classification

media_type
Returns the just the media type 'type/subtype' without any paramters (read-only).

minor
Minor media sub-classification

set(*content_type_string*, *with_parameters=True*)
Parses the content type string and sets this object to it's value.

For a more complete description of the arguments, see the documentation for the parse_media_type() function in this module.

set_parameters(*parameter_list_or_dict*)
Sets the optional paramters based upon the parameter list.

The paramter list should be a semicolon-separated name=value string. Any paramters which already exist on this object will be deleted, unless they appear in the given paramter_list.

rdflib.plugins.parsers.pyRdfa.extras.httpheader.**http_datetime**(*dt=None*)
Formats a datetime as an HTTP 1.1 Date/Time string.

Takes a standard Python datetime object and returns a string formatted according to the HTTP 1.1 date/time format.

If no datetime is provided (or None) then the current time is used.

ABOUT TIMEZONES: If the passed in datetime object is naive it is assumed to be in UTC already. But if it has a tzinfo component, the returned timestamp string will have been converted to UTC automatically. So if you use timezone-aware datetimes, you need not worry about conversion to UTC.

rdflib.plugins.parsers.pyRdfa.extras.httpheader.**is_token**(*s*)
Determines if the string is a valid token.

class rdflib.plugins.parsers.pyRdfa.extras.httpheader.**language_tag**(*tagname*)
Bases: object

This class represents an RFC 3066 language tag.

Initialize objects of this class with a single string representing the language tag, such as "en-US".

Case is insensitive. Wildcarded subtags are ignored or stripped as they have no significance, so that "en-*" is the same as "en". However the universal wildcard "*" language tag is kept as-is.

Note that although relational operators such as < are defined, they only form a partial order based upon specialization.

Thus for example, "en" <= "en-US"

but, not "en" <= "de", and not "de" <= "en".

__eq__(*other*)
== operator. Are the two languages the same?

__ge__(*other*)
>= operator. Returns True if this language is the same as or a more specialized dialect of the other one.

__gt__(*other*)
> operator. Returns True if this language is a more specialized dialect of the other one.

__init__(*tagname*)
Initialize objects of this class with a single string representing the language tag, such as "en-US". Case is insensitive.

__le__(*other*)
<= operator. Returns True if the other language is the same as or a more specialized dialect of this one.

__len__()
Number of subtags in this tag.

__lt__(*other*)
< operator. Returns True if the other language is a more specialized dialect of this one.

__module__ = 'rdflib.plugins.parsers.pyRdfa.extras.httpheader'

__neq__(*other*)
!= operator. Are the two languages different?

__repr__()
The python representation of this language tag.

__str__()
The standard string form of this language tag.

__unicode__()
The unicode string form of this language tag.

all_superiors(*include_wildcard=False*)
Returns a list of this language and all it's superiors.

If include_wildcard is False, then "*" will not be among the output list, unless this language is itself "*".

dialect_of(*other*, *ignore_wildcard=True*)
Is this language a dialect (or subset/specialization) of another.

This method returns True if this language is the same as or a specialization (dialect) of the other language_tag.

If ignore_wildcard is False, then all languages will be considered to be a dialect of the special language tag of "*".

is_universal_wildcard()
Returns True if this language tag represents all possible languages, by using the reserved tag of "*".

superior()
Returns another instance of language_tag which is the superior.

Thus en-US gives en, and en gives *.

rdflib.plugins.parsers.pyRdfa.extras.httpheader.**parse_accept_header**(*header_value*)
Parses the Accept: header.

The value of the header as a string should be passed in; without the header name itself.

This will parse the value of any of the HTTP headers "Accept", "Accept-Charset", "Accept-Encoding", or "Accept-Language". These headers are similarly formatted, in that they are a list of items with associated quality factors. The quality factor, or qvalue, is a number in the range [0.0..1.0] which indicates the relative preference of each item.

This function returns a list of those items, sorted by preference (from most-prefered to least-prefered). Each item in the returned list is actually a tuple consisting of:

(item_name, item_parms, qvalue, accept_parms)

As an example, the following string, text/plain; charset="utf-8"; q=.5; columns=80

would be parsed into this resulting tuple, ('text/plain', [('charset','utf-8')], 0.5, [('columns','80')])

The value of the returned item_name depends upon which header is being parsed, but for example it may be a MIME content or media type (without parameters), a language tag, or so on. Any optional parameters (delimited by semicolons) occuring before the "q=" attribute will be in the item_parms list as (attribute,value) tuples in the same order as they appear in the header. Any quoted values will have been unquoted and unescaped.

The qvalue is a floating point number in the inclusive range 0.0 to 1.0, and roughly indicates the preference for this item. Values outside this range will be capped to the closest extreme.

(!) Note that a qvalue of 0 indicates that the item is explicitly NOT acceptable to the user agent, and should be handled differently by the caller.

The accept_parms, like the item_parms, is a list of any attributes occuring after the "q=" attribute, and will be in the list as (attribute,value) tuples in the same order as they occur. Usually accept_parms will be an empty list, as the HTTP spec allows these extra parameters in the syntax but does not currently define any possible values.

All empty items will be removed from the list. However, duplicate or conflicting values are not detected or handled in any way by this function.

rdflib.plugins.parsers.pyRdfa.extras.httpheader.**parse_accept_language_header**(*header_value*)
Parses the Accept-Language header.

Returns a list of tuples, each like:

(language_tag, qvalue, accept_parameters)

rdflib.plugins.parsers.pyRdfa.extras.httpheader.**parse_comma_list**(*s*, *start=0*, *element_parser=None*, *min_count=0*, *max_count=0*)

Parses a comma-separated list with optional whitespace.

Takes an optional callback function *element_parser*, which is assumed to be able to parse an individual element. It will be passed the string and a *start* argument, and is expected to return a tuple (parsed_result, chars_consumed).

If no element_parser is given, then either single tokens or quoted strings will be parsed.

If min_count > 0, then at least that many non-empty elements must be in the list, or an error is raised.

If max_count > 0, then no more than that many non-empty elements may be in the list, or an error is raised.

rdflib.plugins.parsers.pyRdfa.extras.httpheader.**parse_comment**(*s*, *start=0*)

Parses a ()-style comment from a header value.

Returns tuple (comment, chars_consumed), where the comment will have had the outer-most parentheses and white space stripped. Any nested comments will still have their parentheses and whitespace left intact.

All -escaped quoted pairs will have been replaced with the actual characters they represent, even within the inner nested comments.

You should note that only a few HTTP headers, such as User-Agent or Via, allow ()-style comments within the header value.

A comment is defined by RFC 2616 section 2.2 as:

comment = “(” *(ctext | quoted-pair | comment) ”)” ctext = <any TEXT excluding “(” and ”)”>

rdflib.plugins.parsers.pyRdfa.extras.httpheader.**parse_http_datetime**(*datestring*, *utc_tzinfo=None*, *strict=False*)

Returns a datetime object from an HTTP 1.1 Date/Time string.

Note that HTTP dates are always in UTC, so the returned datetime object will also be in UTC.

You can optionally pass in a tzinfo object which should represent the UTC timezone, and the returned datetime will then be timezone-aware (allowing you to more easly translate it into different timzeones later).

If you set ‘strict’ to True, then only the RFC 1123 format is recognized. Otherwise the backwards-compatible RFC 1036 and Unix asctime(3) formats are also recognized.

Please note that the day-of-the-week is not validated. Also two-digit years, although not HTTP 1.1 compliant, are treated according to recommended Y2K rules.

rdflib.plugins.parsers.pyRdfa.extras.httpheader.**parse_media_type**(*media_type*, *start=0*, *with_parameters=True*)

Parses a media type (MIME type) designator into it’s parts.

Given a media type string, returns a nested tuple of it’s parts.

((major,minor,parmlist), chars_consumed)

where parmlist is a list of tuples of (parm_name, parm_value). Quoted-values are appropriately unquoted and unescaped.

If ‘with_parameters’ is False, then parsing will stop immediately after the minor media type; and will not proceed to parse any of the semicolon-separated paramters.

Examples: image/png -> ((‘image’,’png’,[]), 9) text/plain; charset=”utf-16be”

-> ((‘text’,’plain’,[(‘charset,’utf-16be’)]), 30)

rdflib.plugins.parsers.pyRdfa.extras.httpheader.**parse_number**(*s*, *start=0*)
Parses a positive decimal integer number from the string.

A tuple is returned (number, chars_consumed). If the string is not a valid decimal number, then (None,0) is returned.

rdflib.plugins.parsers.pyRdfa.extras.httpheader.**parse_parameter_list**(*s*, *start=0*)
Parses a semicolon-separated ‘parameter=value’ list.

Returns a tuple (parmlist, chars_consumed), where parmlist is a list of tuples (parm_name, parm_value).

The parameter values will be unquoted and unescaped as needed.

Empty parameters (as in ”;;”) are skipped, as is insignificant white space. The list returned is kept in the same order as the parameters appear in the string.

rdflib.plugins.parsers.pyRdfa.extras.httpheader.**parse_quoted_string**(*s*, *start=0*)
Parses a quoted string.

Returns a tuple (string, chars_consumed). The quote marks will have been removed and all -escapes will have been replaced with the characters they represent.

rdflib.plugins.parsers.pyRdfa.extras.httpheader.**parse_qvalue_accept_list**(*s*, *start=0*, *item_parser=<function parse_token>*)
Parses any of the Accept-* style headers with quality factors.

This is a low-level function. It returns a list of tuples, each like: (item, item_parms, qvalue, accept_parms)

You can pass in a function which parses each of the item strings, or accept the default where the items must be simple tokens. Note that your parser should not consume any paramters (past the special “q” paramter anyway).

The item_parms and accept_parms are each lists of (name,value) tuples.

The qvalue is the quality factor, a number from 0 to 1 inclusive.

rdflib.plugins.parsers.pyRdfa.extras.httpheader.**parse_range_header**(*header_value*, *valid_units=(‘bytes’, ‘none’)*)
Parses the value of an HTTP Range: header.

The value of the header as a string should be passed in; without the header name itself.

Returns a range_set object.

rdflib.plugins.parsers.pyRdfa.extras.httpheader.**parse_range_set**(*s*, *start=0*, *valid_units=(‘bytes’, ‘none’)*)
Parses a (byte) range set specifier.

Returns a tuple (range_set, chars_consumed).

rdflib.plugins.parsers.pyRdfa.extras.httpheader.**parse_range_spec**(*s*, *start=0*)
Parses a (byte) range_spec.

Returns a tuple (range_spec, chars_consumed).

rdflib.plugins.parsers.pyRdfa.extras.httpheader.**parse_token**(*s*, *start=0*)
Parses a token.

A token is a string defined by RFC 2616 section 2.2 as: token = 1*<any CHAR except CTLs or separators>

Returns a tuple (token, chars_consumed), or (‘’,0) if no token starts at the given string position. On a syntax error, a ParseError exception will be raised.

rdflib.plugins.parsers.pyRdfa.extras.httpheader.**parse_token_or_quoted_string**(*s*, *start=0*, *allow_quoted=True*, *allow_token=True*)

Parses a token or a quoted-string.

‘s’ is the string to parse, while start is the position within the string where parsing should begin. It will returns a tuple (token, chars_consumed), with all -escapes and quotation already processed.

Syntax is according to BNF rules in RFC 2161 section 2.2, specifically the ‘token’ and ‘quoted-string’ declarations. Syntax errors in the input string will result in ParseError being raised.

If allow_quoted is False, then only tokens will be parsed instead of either a token or quoted-string.

If allow_token is False, then only quoted-strings will be parsed instead of either a token or quoted-string.

rdflib.plugins.parsers.pyRdfa.extras.httpheader.**quote_string**(*s*, *always_quote=True*)

Produces a quoted string according to HTTP 1.1 rules.

If always_quote is False and if the string is also a valid token, then this function may return a string without quotes.

class rdflib.plugins.parsers.pyRdfa.extras.httpheader.**range_set**

Bases: object

A collection of range_specs, with units (e.g., bytes).

__init__()

__module__ = ‘rdflib.plugins.parsers.pyRdfa.extras.httpheader’

__repr__()

__slots__ = [’units’, ‘range_specs’]

__str__()

coalesce()

Collapses all consecutive range_specs which together define a contiguous range.

Note though that this method will not re-sort the range_specs, so a potentially contiguous range may not be collapsed if they are not sorted. For example the ranges:

10-20, 30-40, 20-30

will not be collapsed to just 10-40. However if the ranges are sorted first as with:

10-20, 20-30, 30-40

then they will collapse to 10-40.

fix_to_size(*size*)

Changes all length-relative range_specs to absolute range_specs based upon given file size. If none of the range_specs in this set can be satisfied, then the entire set is considered unsatifiable and an error is raised. Otherwise any unsatisfiable range_specs will simply be removed from this set.

from_str(*s*, *valid_units=(‘bytes’, ‘none’)*)

Sets this range set based upon a string, such as the Range: header.

You can also use the parse_range_set() function for more control.

If a parsing error occurs, the pre-exising value of this range set is left unchanged.

is_contiguous()
Can the collection of range_specs be coalesced into a single contiguous range?

is_single_range()
Does this range specifier consist of only a single range set?

range_specs

units

class rdflib.plugins.parsers.pyRdfa.extras.httpheader.**range_spec**(*first=0*, *last=None*)

Bases: object

A single contiguous (byte) range.

A range_spec defines a range (of bytes) by specifying two offsets, the 'first' and 'last', which are inclusive in the range. Offsets are zero-based (the first byte is offset 0). The range can not be empty or negative (has to satisfy first <= last).

The range can be unbounded on either end, represented here by the None value, with these semantics:

- A 'last' of None always indicates the last possible byte

  (although that offset may not be known).

- A 'first' of None indicates this is a suffix range, where the last value is actually interpreted to be the number of bytes at the end of the file (regardless of file size).

Note that it is not valid for both first and last to be None.

__contains__(*offset*)
Does this byte range contain the given byte offset?

If the offset < 0, then it is taken as an offset from the end of the file, where -1 is the last byte. This type of offset will only work with suffix ranges.

__eq__(*other*)
Compare ranges for equality.

Note that if non-specific ranges are involved (such as 34- and -5), they could compare as not equal even though they may represent the same set of bytes in some contexts.

__ge__(*other*)
>= operator is not defined

__gt__(*other*)
> operator is not defined

__init__(*first=0*, *last=None*)

__le__(*other*)
<= operator is not defined

__lt__(*other*)
< operator is not defined

__module__ = 'rdflib.plugins.parsers.pyRdfa.extras.httpheader'

__ne__(*other*)
Compare ranges for inequality.

Note that if non-specific ranges are involved (such as 34- and -5), they could compare as not equal even though they may represent the same set of bytes in some contexts.

__repr__()

__slots__ = ['first', 'last']

__str__()
Returns a string form of the range as would appear in a Range: header.

copy()
Makes a copy of this range object.

first

fix_to_size(*size*)
Changes a length-relative range to an absolute range based upon given file size.

Ranges that are already absolute are left as is.

Note that zero-length files are handled as special cases, since the only way possible to specify a zero-length range is with the suffix range "-0". Thus unless this range is a suffix range, it can not satisfy a zero-length file.

If the resulting range (partly) lies outside the file size then an error is raised.

is_fixed()
Returns True if this range is absolute and a fixed size.

This occurs only if neither first or last is None. Converse is the is_unbounded() method.

is_suffix()
Returns True if this is a suffix range.

A suffix range is one that specifies the last N bytes of a file regardless of file size.

is_unbounded()
Returns True if the number of bytes in the range is unspecified.

This can only occur if either the 'first' or the 'last' member is None. Converse is the is_fixed() method.

is_whole_file()
Returns True if this range includes all possible bytes.

This can only occur if the 'last' member is None and the first member is 0.

last

merge_with(*other*)
Tries to merge the given range into this one.

The size of this range may be enlarged as a result.

An error is raised if the two ranges do not overlap or are not contiguous with each other.

set(*first*, *last*)
Sets the value of this range given the first and last offsets.

rdflib.plugins.parsers.pyRdfa.extras.httpheader.**remove_comments**(*s*, *collapse_spaces=True*)

Removes any ()-style comments from a string.

In HTTP, ()-comments can nest, and this function will correctly deal with that.

If 'collapse_spaces' is True, then if there is any whitespace surrounding the comment, it will be replaced with a single space character. Whitespace also collapses across multiple comment sequences, so that "a (b) (c) d" becomes just "a d".

Otherwise, if 'collapse_spaces' is False then all whitespace which is outside any comments is left intact as-is.

host Package

host Package Host language sub-package for the pyRdfa package. It contains variables and possible modules necessary to manage various RDFa host languages.

This module may have to be modified if a new host language is added to the system. In many cases the rdfa_core as a host language is enough, because there is no need for a special processing. However, some host languages may require an initial context, or their value may control some transformations, in which case additional data have to be added to this module. This module header contains all tables and arrays to be adapted, and the module content may contain specific transformation methods.

@summary: RDFa Host package @requires: U{RDFLib package<http://rdflib.net>} @organization: U{World Wide Web Consortium<http://www.w3.org>} @author: U{Ivan Herman<a href="http://www.w3.org/People/Ivan/">} @license: This software is available for use under the U{W3C® SOFTWARE NOTICE AND LICENSE<href="http://www.w3.org/Consortium/Legal/2002/copyright-software-20021231">}

@var content_to_host_language: a dictionary mapping a media type to a host language @var preferred_suffixes: mapping from preferred suffixes for media types; used if the file is local, ie, there is not HTTP return value for the media type. It corresponds to the preferred suffix in the media type registration @var initial_contexts: mapping from host languages to list of initial contexts @var accept_xml_base: list of host languages that accept the xml:base attribute for base setting @var accept_xml_lang: list of host languages that accept the xml:lang attribute for language setting. Note that XHTML and HTML have some special rules, and those are hard coded... @var warn_xmlns_usage: list of host languages that should generate a warning for the usage of @xmlns (for RDFa 1.1) @var accept_embedded_rdf_xml: list of host languages that might also include RDF data using an embedded RDF/XML (e.g., SVG). That RDF data may be merged with the output @var accept_embedded_turtle: list of host languages that might also include RDF data using a C{script} element. That RDF data may be merged with the output @var require_embedded_rdf: list of languages that must accept embedded RDF, ie, the corresponding option is irrelevant @var host_dom_transforms: dictionary mapping a host language to an array of methods that are invoked at the beginning of the parsing process for a specific node. That function can do a last minute change on that DOM node, eg, adding or modifying an attribute. The method's signature is (node, state), where node is the DOM node, and state is the L{Execution context<pyRdfa.state.ExecutionContext>}. @var predefined_1_0_rel: terms that are hardcoded for HTML+RDF1.0 and replace the initial context for that version @var beautifying_prefixes: this is really just to make the output more attractive: for each media type a dictionary of prefix-URI pairs that can be used to make the terms look better... @var default_vocabulary: as its name suggests, default @vocab value for a specific host language

class rdflib.plugins.parsers.pyRdfa.host.**HostLanguage**
An enumeration style class: recognized host language types for this processor of RDFa. Some processing details may depend on these host languages. "rdfa_core" is the default Host Language is nothing else is defined.

__module__ = 'rdflib.plugins.parsers.pyRdfa.host'

atom = 'Atom+RDFa'

html5 = 'HTML5+RDFa'

rdfa_core = 'RDFa Core'

svg = 'SVG+RDFa'

xhtml = 'XHTML+RDFa'

xhtml5 = 'XHTML5+RDFa'

class rdflib.plugins.parsers.pyRdfa.host.**MediaTypes**
An enumeration style class: some common media types (better have them at one place to avoid misstyping...)

__module__ = 'rdflib.plugins.parsers.pyRdfa.host'

atom = 'application/atom+xml'

html = 'text/html'

nt = 'text/plain'

rdfxml = 'application/rdf+xml'

smil = 'application/smil+xml'

svg = 'application/svg+xml'

svgi = 'image/svg+xml'

turtle = 'text/turtle'

xhtml = 'application/xhtml+xml'

xml = 'application/xml'

xmlt = 'text/xml'

rdflib.plugins.parsers.pyRdfa.host.**adjust_html_version**(*input*, *rdfa_version*)

Adjust the rdfa_version based on the (possible) DTD @param input: the input stream that has to be parsed by an xml parser @param rdfa_version: the current rdfa_version; will be returned if nothing else is found @return: the rdfa_version, either "1.0" or "1.1, if the DTD says so, otherwise the input rdfa_version value

rdflib.plugins.parsers.pyRdfa.host.**adjust_xhtml_and_version**(*dom*, *incoming_language*, *rdfa_version*)

Check if the xhtml+RDFa is really XHTML 0 or 1 or whether it should be considered as XHTML5. This is done by looking at the DTD. Furthermore, checks whether whether the system id signals an rdfa 1.0, in which case the version is also set.

@param dom: top level DOM node @param incoming_language: host language to be checked; the whole check is relevant for xhtml only. @param rdfa_version: rdfa_version as known by the caller @return: a tuple of the possibly modified host language (ie, set to XHTML5) and the possibly modified rdfa version (ie, set to "1.0", "1.1", or the incoming rdfa_version if nothing is found)

atom Module Simple transfomer for Atom: the C{@typeof=""} is added to the C{<entry>} element (unless something is already there).

@summary: Add a top "about" to <head> and <body> @requires: U{RDFLib package<http://rdflib.net>} @organization: U{World Wide Web Consortium<http://www.w3.org>} @author: U{Ivan Herman<a href="http://www.w3.org/People/Ivan/">} @license: This software is available for use under the U{W3C® SOFTWARE NOTICE AND LICENSE<href="http://www.w3.org/Consortium/Legal/2002/copyright-software-20021231">} @contact: Ivan Herman, ivan@w3.org

rdflib.plugins.parsers.pyRdfa.host.atom.**atom_add_entry_type**(*node*, *state*)

@param node: the current node that could be modified @param state: current state @type state: L{Execution context<pyRdfa.state.ExecutionContext>}

html5 Module Simple transfomer for HTML5: add a @src for any @data, add a @content for the @value attribute of the <data> element, and interpret the <time> element.

@summary: Add a top "about" to <head> and <body> @requires: U{RDFLib package<http://rdflib.net>} @organization: U{World Wide Web Consortium<http://www.w3.org>} @author: U{Ivan Herman<a href="http://www.w3.org/People/Ivan/">} @license: This software is available for use under the U{W3C® SOFTWARE NOTICE AND LICENSE<href="http://www.w3.org/Consortium/Legal/2002/copyright-software-20021231">} @contact: Ivan Herman, ivan@w3.org

rdflib.plugins.parsers.pyRdfa.host.html5.**html5_extra_attributes**(*node*, *state*)
@param node: the current node that could be modified @param state: current state @type state: L{Execution context<pyRdfa.state.ExecutionContext>}

rdflib.plugins.parsers.pyRdfa.host.html5.**remove_rel**(*node*, *state*)
If @property and @rel/@rev are on the same element, then only CURIE and URI can appear as a rel/rev value.

@param node: the current node that could be modified @param state: current state @type state: L{Execution context<pyRdfa.state.ExecutionContext>}

rdfs Package

rdfs Package Separate module to handle vocabulary expansions. The L{cache} module takes care of caching vocabulary graphs; the L{process} module takes care of the expansion itself.

@organization: U{World Wide Web Consortium<http://www.w3.org>} @author: U{Ivan Herman<a href="http://www.w3.org/People/Ivan/">} @license: This software is available for use under the U{W3C® SOFTWARE NOTICE AND LICENSE<href="http://www.w3.org/Consortium/Legal/2002/copyright-software-20021231">}

cache Module Managing Vocab Caching.

@summary: RDFa parser (distiller) @requires: U{RDFLib<http://rdflib.net>} @organization: U{World Wide Web Consortium<http://www.w3.org>} @author: U{Ivan Herman<a href="http://www.w3.org/People/Ivan/">} @license: This software is available for use under the U{W3C® SOFTWARE NOTICE AND LICENSE<href="http://www.w3.org/Consortium/Legal/2002/copyright-software-20021231">}

class rdflib.plugins.parsers.pyRdfa.rdfs.cache.**CachedVocab**(*URI*, *options=None*)
Bases: *rdflib.plugins.parsers.pyRdfa.rdfs.cache.CachedVocabIndex*

Cache for a specific vocab. The content of the cache is the graph. These are also the data that are stored on the disc (in pickled form)

@ivar graph: the RDF graph @ivar URI: vocabulary URI @ivar filename: file name (not the complete path) of the cached version @ivar creation_date: creation date of the cache @type creation_date: datetime @ivar expiration_date: expiration date of the cache @type expiration_date: datetime @cvar runtime_cache : a run time cache for already 'seen' vocabulary files. Apart from (marginally) speeding up processing, this also prevents recursion @type runtime_cache : dictionary

__init__(*URI*, *options=None*)
@param URI: real URI for the vocabulary file @param options: the error handler (option) object to send warnings to @type options: L{options.Options}

__module__ = 'rdflib.plugins.parsers.pyRdfa.rdfs.cache'

class rdflib.plugins.parsers.pyRdfa.rdfs.cache.**CachedVocabIndex**(*options=None*)
Class to manage the cache index. Takes care of finding the vocab directory, and manages the index to the individual vocab data.

The vocab directory is set to a platform specific area, unless an environment variable sets it explicitly. The environment variable is "PyRdfaCacheDir"

Every time the index is changed, the index is put back (via pickle) to the directory.

@ivar app_data_dir: directory for the vocabulary cache directory @ivar index_fname: the full path of the index file on the disc @ivar indeces: the in-memory version of the index (a directory mapping URI-s to tuples) @ivar options: the error handler (option) object to send warnings to @type options: L{options.Options} @ivar report: whether details on the caching should be reported @type report: Boolean @cvar vocabs: File name used for the

index in the cache directory @cvar preference_path: Cache directories for the three major platforms (ie, mac, windows, unix) @type preference_path: directory, keyed by “mac”, “win”, and “unix” @cvar architectures: Various ‘architectures’ as returned by the python call, and their mapping on one of the major platforms. If an architecture is missing, it is considered to be “unix” @type architectures: directory, mapping architectures to “mac”, “win”, or “unix”

__init__(*options=None*)
@param options: the error handler (option) object to send warnings to @type options: L{options.Options}

__module__ = ‘rdflib.plugins.parsers.pyRdfa.rdfs.cache’

add_ref(*uri*, *vocab_reference*)
Add a new entry to the index, possibly removing the previous one.

@param uri: the URI that serves as a key in the index directory @param vocab_reference: tuple consisting of file name, modification date, and expiration date

architectures = {‘win32’: ‘win’, ‘darwin’: ‘mac’, ‘nt’: ‘win’, ‘cygwin’: ‘win’}

get_ref(*uri*)
Get an index entry, if available, None otherwise. The return value is a tuple: file name, modification date, and expiration date

@param uri: the URI that serves as a key in the index directory

preference_path = {‘win’: ‘pyRdfa-cache’, ‘mac’: ‘Library/Application Support/pyRdfa-cache’, ‘unix’: ‘.pyRdfa-cac

vocabs = ‘cache_index’

rdflib.plugins.parsers.pyRdfa.rdfs.cache.**offline_cache_generation**(*args*)
Generate a cache for the vocabulary in args.

@param args: array of vocabulary URIs.

process Module @organization: U{World Wide Web Consortium<http://www.w3.org>} @author: U{Ivan Herman<a href=”http://www.w3.org/People/Ivan/“>} @license: This software is available for use under the U{W3C® SOFTWARE NOTICE AND LICENSE<href=”http://www.w3.org/Consortium/Legal/2002/copyright-software-20021231“>}

class rdflib.plugins.parsers.pyRdfa.rdfs.process.**MiniOWL**(*graph*, *schema_semantics=False*)
Class implementing the simple OWL RL Reasoning required by RDFa in managing vocabulary files. This is done via a forward chaining process (in the L{closure} method) using a few simple rules as defined by the RDF and the OWL Semantics specifications.

@ivar graph: the graph that has to be expanded @ivar added_triples: each cycle collects the triples that are to be added to the graph eventually. @type added_triples: a set, to ensure the unicity of triples being added

__init__(*graph*, *schema_semantics=False*)

__module__ = ‘rdflib.plugins.parsers.pyRdfa.rdfs.process’

closure()
Generate the closure the graph. This is the real ‘core’.

The processing rules store new triples via the L{separate method<store_triple>} which stores them in the L{added_triples<added_triples>} array. If that array is emtpy at the end of a cycle, it means that the whole process can be stopped.

rules(*t*)
Go through the OWL-RL entailement rules prp-spo1, prp-eqp1, prp-eqp2, cax-sco, cax-eqc1, and cax-eqc2 by extending the graph. @param t: a triple (in the form of a tuple)

store_triple(*t*)

In contrast to its name, this does not yet add anything to the graph itself, it just stores the tuple in an L{internal set<added_triples>}. (It is important for this to be a set: some of the rules in the various closures may generate the same tuples several times.) Before adding the tuple to the set, the method checks whether the tuple is in the final graph already (if yes, it is not added to the set).

The set itself is emptied at the start of every processing cycle; the triples are then effectively added to the graph at the end of such a cycle. If the set is actually empty at that point, this means that the cycle has not added any new triple, and the full processing can stop.

@param t: the triple to be added to the graph, unless it is already there @type t: a 3-element tuple of (s,p,o)

rdflib.plugins.parsers.pyRdfa.rdfs.process.**process_rdfa_sem**(*graph*, *options*)

Expand the graph through the minimal RDFS and OWL rules defined for RDFa.

The expansion is done in several steps:

1. the vocabularies are retrieved from the incoming graph (there are RDFa triples generated for that)
2. all vocabularies are merged into a separate vocabulary graph
3. the RDFS/OWL expansion is done on the vocabulary graph, to take care of all the subproperty, subclass, etc, chains
4. the (expanded) vocabulary graph content is added to the incoming graph
5. the incoming graph is expanded
6. the triples appearing in the vocabulary graph are removed from the incoming graph, to avoid unnecessary extra triples from the data

@param graph: an RDFLib Graph instance, to be expanded @param options: options as defined for the RDFa run; used to generate warnings @type options: L{pyRdfa.Options}

rdflib.plugins.parsers.pyRdfa.rdfs.process.**return_graph**(*uri*, *options*, *newCache=False*)

Parse a file, and return an RDFLib Graph. The URI's content type is checked and either one of RDFLib's parsers is invoked (for the Turtle, RDF/XML, and N Triple cases) or a separate RDFa processing is invoked on the RDFa content.

The Accept header of the HTTP request gives a preference to Turtle, followed by RDF/XML and then HTML (RDFa), in case content negotiation is used.

This function is used to retreive the vocabulary file and turn it into an RDFLib graph.

@param uri: URI for the graph @param options: used as a place where warnings can be sent @param newCache: in case this is used with caching, whether a new cache is generated; that modifies the warning text @return: A tuple consisting of an RDFLib Graph instance and an expiration date); None if the dereferencing or the parsing was unsuccessful

transform Package

transform Package Transformer sub-package for the pyRdfa package. It contains modules with transformer functions; each may be invoked by pyRdfa to transform the dom tree before the "real" RDfa processing.

@summary: RDFa Transformer package @requires: U{RDFLib package<http://rdflib.net>} @organization: U{World Wide Web Consortium<http://www.w3.org>} @author: U{Ivan Herman<a href="http://www.w3.org/People/Ivan/">} @license: This software is available for use under the U{W3C® SOFTWARE NOTICE AND LICENSE<href="http://www.w3.org/Consortium/Legal/2002/copyright-software-20021231">}

rdflib.plugins.parsers.pyRdfa.transform.**empty_safe_curie**(*node*, *options*, *state*)
Remove the attributes whose value is an empty safe curie. It also adds an 'artificial' flag, ie, an attribute (called 'emptysc') into the node to signal that there _is_ an attribute with an ignored safe curie value. The name of the attribute is 'about_pruned' or 'resource_pruned'.

@param node: a DOM node for the top level element @param options: invocation options @type options: L{Options<pyRdfa.options>} @param state: top level execution state @type state: L{State<pyRdfa.state>}

rdflib.plugins.parsers.pyRdfa.transform.**top_about**(*root*, *options*, *state*)
@param root: a DOM node for the top level element @param options: invocation options @type options: L{Options<pyRdfa.options>} @param state: top level execution state @type state: L{State<pyRdfa.state>}

rdflib.plugins.parsers.pyRdfa.transform.**vocab_for_role**(*node*, *options*, *state*)
The value of the @role attribute (defined separately in the U{Role Attribute Specification Lite<http://www.w3.org/TR/role-attribute/#using-role-in-conjunction-with-rdfa>}) should be as if a @vocab value to the XHTML vocabulary was defined for it. This method turns all terms in role attributes into full URI-s, so that this would not be an issue for the run-time.

@param node: a DOM node for the top level element @param options: invocation options @type options: L{Options<pyRdfa.options>} @param state: top level execution state @type state: L{State<pyRdfa.state>}

DublinCore Module Transfomer: handles the Dublin Core recommendation for XHTML for adding DC values. What this means is that:

- DC namespaces are defined via C{<link rel="schema.XX" value="...."/>}
- The 'XX.term' is used much like QNames in C{<link>} and C{<meta>} elements. For the latter, the namespaced names are added to a C{@property} attribute.

This transformer adds "real" namespaces and changes the DC references in link and meta elements to abide to the RDFa namespace syntax.

@summary: Dublin Core transformer @requires: U{RDFLib package<http://rdflib.net>} @organization: U{World Wide Web Consortium<http://www.w3.org>} @author: U{Ivan Herman<a href="http://www.w3.org/People/Ivan/">} @license: This software is available for use under the U{W3C® SOFTWARE NOTICE AND LICENSE<href="http://www.w3.org/Consortium/Legal/2002/copyright-software-20021231">} @contact: Ivan Herman, ivan@w3.org

rdflib.plugins.parsers.pyRdfa.transform.DublinCore.**DC_transform**(*html*, *options*, *state*)
@param html: a DOM node for the top level html element @param options: invocation options @type options: L{Options<pyRdfa.options>} @param state: top level execution state @type state: L{State<pyRdfa.state>}

OpenID Module Simple transfomer: handle OpenID elements. Ie: an openid namespace is added and the usual 'link' elements for openid are exchanged against a namespaced version.

@summary: OpenID transformer module. @requires: U{RDFLib package<http://rdflib.net>} @organization: U{World Wide Web Consortium<http://www.w3.org>} @author: U{Ivan Herman<a href="http://www.w3.org/People/Ivan/">} @license: This software is available for use under the U{W3C® SOFTWARE NOTICE AND LICENSE<href="http://www.w3.org/Consortium/Legal/2002/copyright-software-20021231">} @contact: Ivan Herman, ivan@w3.org @var OPENID_NS: the OpenID URI used in the package

rdflib.plugins.parsers.pyRdfa.transform.OpenID.**OpenID_transform**(*html*, *options*, *state*)
Replace C{openid.XXX} type C{@rel} attribute values in C{<link>} elements by C{openid:XXX}. The openid URI is also added to the top level namespaces with the C{openid:} local name.

@param html: a DOM node for the top level html element @param options: invocation options @type options: L{Options<pyRdfa.options>} @param state: top level execution state @type state: L{State<pyRdfa.state>}

lite Module @author: U{Ivan Herman<a href="http://www.w3.org/People/Ivan/">} @license: This software is available for use under the U{W3C® SOFTWARE NOTICE AND LICENSE<href="http://www.w3.org/Consortium/Legal/2002/copyright-software-20021231">} @contact: Ivan Herman, ivan@w3.org @version: $Id: lite.py,v 1.11 2013-09-26 16:37:54 ivan Exp $ $Date: 2013-09-26 16:37:54 $

rdflib.plugins.parsers.pyRdfa.transform.lite.**lite_prune**(*top*, *options*, *state*)

This is a misnomer. The current version does not remove anything from the tree, just generates warnings as for the usage of non-lite attributes. A more aggressive version would mean to remove those attributes, but that would, in fact, define an RDFa Lite conformance level in the parser, which is against the WG decisions. So this should not be done; the corresponding commands are commented in the code below...

@param top: a DOM node for the top level element @param options: invocation options @type options: L{Options<pyRdfa.options>} @param state: top level execution state @type state: L{State<pyRdfa.state>}

metaname Module Simple transfomer: C{meta} element is extended with a C{property} attribute, with a copy of the C{name} attribute values.

@author: U{Ivan Herman<a href="http://www.w3.org/People/Ivan/">} @license: This software is available for use under the U{W3C® SOFTWARE NOTICE AND LICENSE<href="http://www.w3.org/Consortium/Legal/2002/copyright-software-20021231">} @contact: Ivan Herman, ivan@w3.org @version: $Id: metaname.py,v 1.3 2012-01-18 14:16:45 ivan Exp $ $Date: 2012-01-18 14:16:45 $

rdflib.plugins.parsers.pyRdfa.transform.metaname.**meta_transform**(*html*, *options*, *state*)

@param html: a DOM node for the top level html element @param options: invocation options @type options: L{Options<pyRdfa.options>} @param state: top level execution state @type state: L{State<pyRdfa.state>}

prototype Module Encoding of the RDFa prototype vocabulary behavior. This means processing the graph by adding and removing triples based on triples using the rdfa:Prototype and rdfa:ref class and property, respectively. For details, see the HTML5+RDFa document.

@author: U{Ivan Herman<a href="http://www.w3.org/People/Ivan/">} @license: This software is available for use under the U{W3C® SOFTWARE NOTICE AND LICENSE<href="http://www.w3.org/Consortium/Legal/2002/copyright-software-20021231">} @contact: Ivan Herman, ivan@w3.org @version: $Id: prototype.py,v 1.1 2013-01-18 09:41:49 ivan Exp $ $Date: 2013-01-18 09:41:49 $

rdflib.plugins.parsers.pyRdfa.transform.prototype.**handle_prototypes**(*graph*)

serializers Package

n3 Module Notation 3 (N3) RDF graph serializer for RDFLib.

class rdflib.plugins.serializers.n3.**N3Serializer**(*store*, *parent=None*)

Bases: *rdflib.plugins.serializers.turtle.TurtleSerializer*

__init__(*store*, *parent=None*)

__module__ = 'rdflib.plugins.serializers.n3'

endDocument()

getQName(*uri*, *gen_prefix=True*)

indent(*modifier=0*)

isDone(*subject*)

p_clause(*node*, *position*)

path(*node*, *position*, *newline=False*)

preprocessTriple(*triple*)

reset()

s_clause(*subject*)

short_name = 'n3'

startDocument()

statement(*subject*)

subjectDone(*subject*)

nquads Module

class rdflib.plugins.serializers.nquads.**NQuadsSerializer**(*store*)

Bases: *rdflib.serializer.Serializer*

__init__(*store*)

__module__ = 'rdflib.plugins.serializers.nquads'

serialize(*stream*, *base=None*, *encoding=None*, ***args*)

nt Module N-Triples RDF graph serializer for RDFLib. See <http://www.w3.org/TR/rdf-testcases/#ntriples> for details about the format.

class rdflib.plugins.serializers.nt.**NTSerializer**(*store*)

Bases: *rdflib.serializer.Serializer*

Serializes RDF graphs to NTriples format.

__module__ = 'rdflib.plugins.serializers.nt'

serialize(*stream*, *base=None*, *encoding=None*, ***args*)

rdfxml Module

rdflib.plugins.serializers.rdfxml.**fix**(*val*)

strip off _: from nodeIDs... as they are not valid NCNames

class rdflib.plugins.serializers.rdfxml.**XMLSerializer**(*store*)

Bases: *rdflib.serializer.Serializer*

__init__(*store*)

__module__ = 'rdflib.plugins.serializers.rdfxml'

predicate(*predicate*, *object*, *depth=1*)

serialize(*stream*, *base=None*, *encoding=None*, ***args*)

subject(*subject*, *depth=1*)

class rdflib.plugins.serializers.rdfxml.**PrettyXMLSerializer**(*store*, *max_depth=3*)

Bases: *rdflib.serializer.Serializer*

__init__(*store*, *max_depth=3*)

__module__ = 'rdflib.plugins.serializers.rdfxml'

predicate(*predicate*, *object*, *depth=1*)

serialize(*stream*, *base=None*, *encoding=None*, ***args*)

subject(*subject*, *depth=1*)

trig Module Trig RDF graph serializer for RDFLib. See <http://www.w3.org/TR/trig/> for syntax specification.

class rdflib.plugins.serializers.trig.**TrigSerializer**(*store*)
Bases: *rdflib.plugins.serializers.turtle.TurtleSerializer*

__init__(*store*)

__module__ = 'rdflib.plugins.serializers.trig'

indentString = u' '

preprocess()

reset()

serialize(*stream*, *base=None*, *encoding=None*, *spacious=None*, ***args*)

short_name = 'trig'

trix Module

class rdflib.plugins.serializers.trix.**TriXSerializer**(*store*)
Bases: *rdflib.serializer.Serializer*

__init__(*store*)

__module__ = 'rdflib.plugins.serializers.trix'

serialize(*stream*, *base=None*, *encoding=None*, ***args*)

turtle Module Turtle RDF graph serializer for RDFLib. See <http://www.w3.org/TeamSubmission/turtle/> for syntax specification.

class rdflib.plugins.serializers.turtle.**RecursiveSerializer**(*store*)
Bases: *rdflib.serializer.Serializer*

__init__(*store*)

__module__ = 'rdflib.plugins.serializers.turtle'

addNamespace(*prefix*, *uri*)

buildPredicateHash(*subject*)
Build a hash key by predicate to a list of objects for the given subject

checkSubject(*subject*)
Check to see if the subject should be serialized yet

indent(*modifier=0*)
Returns indent string multiplied by the depth

indentString = u' '

isDone(*subject*)
Return true if subject is serialized

maxDepth = 10

orderSubjects()

predicateOrder = [rdflib.term.URIRef(u'http://www.w3.org/1999/02/22-rdf-syntax-ns#type'), rdflib.term.URIRef(u'ht

preprocess()

preprocessTriple((*s*, *p*, *o)*)

reset()

sortProperties(*properties*)
Take a hash from predicate uris to lists of values. Sort the lists of values. Return a sorted list of properties.

subjectDone(*subject*)
Mark a subject as done.

topClasses = [rdflib.term.URIRef(u'http://www.w3.org/2000/01/rdf-schema#Class')]

write(*text*)
Write text in given encoding.

class rdflib.plugins.serializers.turtle.**TurtleSerializer**(*store*)
Bases: *rdflib.plugins.serializers.turtle.RecursiveSerializer*

__init__(*store*)

__module__ = 'rdflib.plugins.serializers.turtle'

addNamespace(*prefix*, *namespace*)

doList(*l*)

endDocument()

getQName(*uri*, *gen_prefix=True*)

indentString = ' '

isValidList(*l*)
Checks if l is a valid RDF list, i.e. no nodes have other properties.

label(*node*, *position*)

objectList(*objects*)

p_default(*node*, *position*, *newline=False*)

p_squared(*node*, *position*, *newline=False*)

path(*node*, *position*, *newline=False*)

predicateList(*subject*, *newline=False*)

preprocessTriple(*triple*)

reset()

s_default(*subject*)

s_squared(*subject*)

serialize(*stream*, *base=None*, *encoding=None*, *spacious=None*, ***args*)

short_name = 'turtle'

startDocument()

statement(*subject*)

verb(*node*, *newline=False*)

xmlwriter Module

class rdflib.plugins.serializers.xmlwriter.**XMLWriter**(*stream*, *namespace_manager*, *encoding=None*, *decl=1*, *extra_ns=None*)

Bases: object

__init__(*stream*, *namespace_manager*, *encoding=None*, *decl=1*, *extra_ns=None*)

__module__ = 'rdflib.plugins.serializers.xmlwriter'

attribute(*uri*, *value*)

element(*uri*, *content*, *attributes={}*)
Utility method for adding a complete simple element

indent

namespaces(*namespaces=None*)

pop(*uri=None*)

push(*uri*)

qname(*uri*)
Compute qname for a uri using our extra namespaces, or the given namespace manager

text(*text*)

sparql Package

sparql Package

SPARQL implementation for RDFLib

New in version 4.0.

rdflib.plugins.sparql.**CUSTOM_EVALS = {}**
Custom evaluation functions

These must be functions taking (ctx, part) and raise NotImplementedError if they cannot handle a certain part

rdflib.plugins.sparql.**SPARQL_DEFAULT_GRAPH_UNION = True**
If True - the default graph in the RDF Dataset is the union of all named graphs (like RDFLib's ConjunctiveGraph)

rdflib.plugins.sparql.**SPARQL_LOAD_GRAPHS = True**
If True, using FROM <uri> and FROM NAMED <uri> will load/parse more data

aggregates Module

rdflib.plugins.sparql.aggregates.**agg_Avg**(*a*, *group*, *bindings*)
rdflib.plugins.sparql.aggregates.**agg_Count**(*a*, *group*, *bindings*)
rdflib.plugins.sparql.aggregates.**agg_GroupConcat**(*a*, *group*, *bindings*)
rdflib.plugins.sparql.aggregates.**agg_Max**(*a*, *group*, *bindings*)
rdflib.plugins.sparql.aggregates.**agg_Min**(*a*, *group*, *bindings*)
rdflib.plugins.sparql.aggregates.**agg_Sample**(*a*, *group*, *bindings*)
rdflib.plugins.sparql.aggregates.**agg_Sum**(*a*, *group*, *bindings*)

rdflib.plugins.sparql.aggregates.**evalAgg**(*a*, *group*, *bindings*)

algebra Module Converting the 'parse-tree' output of pyparsing to a SPARQL Algebra expression

http://www.w3.org/TR/sparql11-query/#sparqlQuery

rdflib.plugins.sparql.algebra.**BGP**(*triples=None*)

rdflib.plugins.sparql.algebra.**Extend**(*p*, *expr*, *var*)

rdflib.plugins.sparql.algebra.**Filter**(*expr*, *p*)

rdflib.plugins.sparql.algebra.**Graph**(*term*, *graph*)

rdflib.plugins.sparql.algebra.**Group**(*p*, *expr=None*)

rdflib.plugins.sparql.algebra.**Join**(*p1*, *p2*)

rdflib.plugins.sparql.algebra.**LeftJoin**(*p1*, *p2*, *expr*)

rdflib.plugins.sparql.algebra.**Minus**(*p1*, *p2*)

rdflib.plugins.sparql.algebra.**OrderBy**(*p*, *expr*)

rdflib.plugins.sparql.algebra.**Project**(*p*, *PV*)

exception rdflib.plugins.sparql.algebra.**StopTraversal**(*rv*)
Bases: exceptions.Exception

__init__(*rv*)

__module__ = 'rdflib.plugins.sparql.algebra'

__weakref__
list of weak references to the object (if defined)

rdflib.plugins.sparql.algebra.**ToMultiSet**(*p*)

rdflib.plugins.sparql.algebra.**Union**(*p1*, *p2*)

rdflib.plugins.sparql.algebra.**analyse**(*n*, *children*)

rdflib.plugins.sparql.algebra.**collectAndRemoveFilters**(*parts*)
FILTER expressions apply to the whole group graph pattern in which they appear.

http://www.w3.org/TR/sparql11-query/#sparqlCollectFilters

rdflib.plugins.sparql.algebra.**pprintAlgebra**(*q*)

rdflib.plugins.sparql.algebra.**reorderTriples**(*l*)
Reorder triple patterns so that we execute the ones with most bindings first

rdflib.plugins.sparql.algebra.**simplify**(*n*)
Remove joins to empty BGPs

rdflib.plugins.sparql.algebra.**translate**(*q*)
http://www.w3.org/TR/sparql11-query/#convertSolMod

rdflib.plugins.sparql.algebra.**translateAggregates**(*q*, *M*)

rdflib.plugins.sparql.algebra.**translateExists**(*e*)
Translate the graph pattern used by EXISTS and NOT EXISTS http://www.w3.org/TR/sparql11-query/#sparqlCollectFilters

rdflib.plugins.sparql.algebra.**translateGraphGraphPattern**(*graphPattern*)

rdflib.plugins.sparql.algebra.**translateGroupGraphPattern**(*graphPattern*)
http://www.w3.org/TR/sparql11-query/#convertGraphPattern

rdflib.plugins.sparql.algebra.**translateGroupOrUnionGraphPattern**(*graphPattern*)

rdflib.plugins.sparql.algebra.**translateInlineData**(*graphPattern*)

rdflib.plugins.sparql.algebra.**translatePName**(*p*, *prologue*)
Expand prefixed/relative URIs

rdflib.plugins.sparql.algebra.**translatePath**(*p*)
Translate PropertyPath expressions

rdflib.plugins.sparql.algebra.**translatePrologue**(*p*, *base*, *initNs=None*, *prologue=None*)

rdflib.plugins.sparql.algebra.**translateQuads**(*quads*)

rdflib.plugins.sparql.algebra.**translateQuery**(*q*, *base=None*, *initNs=None*)
Translate a query-parsetree to a SPARQL Algebra Expression

Return a rdflib.plugins.sparql.sparql.Query object

rdflib.plugins.sparql.algebra.**translateUpdate**(*q*, *base=None*, *initNs=None*)
Returns a list of SPARQL Update Algebra expressions

rdflib.plugins.sparql.algebra.**translateUpdate1**(*u*, *prologue*)

rdflib.plugins.sparql.algebra.**translateValues**(*v*)

rdflib.plugins.sparql.algebra.**traverse**(*tree*, *visitPre=<function <lambda>>*, *visitPost=<function <lambda>>*, *complete=None*)
Traverse tree, visit each node with visit function visit function may raise StopTraversal to stop traversal if complete!=None, it is returned on complete traversal, otherwise the transformed tree is returned

rdflib.plugins.sparql.algebra.**triples**(*l*)

compat Module Function/methods to help supporting 2.5-2.7

datatypes Module Utility functions for supporting the XML Schema Datatypes hierarchy

rdflib.plugins.sparql.datatypes.**type_promotion**(*t1*, *t2*)

evaluate Module These method recursively evaluate the SPARQL Algebra

evalQuery is the entry-point, it will setup context and return the SPARQLResult object

evalPart is called on each level and will delegate to the right method

A rdflib.plugins.sparql.sparql.QueryContext is passed along, keeping information needed for evaluation

A list of dicts (solution mappings) is returned, apart from GroupBy which may also return a dict of list of dicts

rdflib.plugins.sparql.evaluate.**evalAggregateJoin**(*ctx*, *agg*)

rdflib.plugins.sparql.evaluate.**evalAskQuery**(*ctx*, *query*)

rdflib.plugins.sparql.evaluate.**evalBGP**(*ctx*, *bgp*)
A basic graph pattern

rdflib.plugins.sparql.evaluate.**evalConstructQuery**(*ctx*, *query*)

rdflib.plugins.sparql.evaluate.**evalDistinct**(*ctx*, *part*)

rdflib.plugins.sparql.evaluate.**evalExtend**(*ctx*, *extend*)

rdflib.plugins.sparql.evaluate.**evalFilter**(*ctx*, *part*)

rdflib.plugins.sparql.evaluate.**evalGraph**(*ctx*, *part*)

rdflib.plugins.sparql.evaluate.**evalGroup**(*ctx*, *group*)
http://www.w3.org/TR/sparql11-query/#defn_algGroup

rdflib.plugins.sparql.evaluate.**evalJoin**(*ctx*, *join*)

rdflib.plugins.sparql.evaluate.**evalLazyJoin**(*ctx*, *join*)
A lazy join will push the variables bound in the first part to the second part, essentially doing the join implicitly hopefully evaluating much fewer triples

rdflib.plugins.sparql.evaluate.**evalLeftJoin**(*ctx*, *join*)

rdflib.plugins.sparql.evaluate.**evalMinus**(*ctx*, *minus*)

rdflib.plugins.sparql.evaluate.**evalMultiset**(*ctx*, *part*)

rdflib.plugins.sparql.evaluate.**evalOrderBy**(*ctx*, *part*)

rdflib.plugins.sparql.evaluate.**evalPart**(*ctx*, *part*)

rdflib.plugins.sparql.evaluate.**evalProject**(*ctx*, *project*)

rdflib.plugins.sparql.evaluate.**evalQuery**(*graph*, *query*, *initBindings*, *base=None*)

rdflib.plugins.sparql.evaluate.**evalReduced**(*ctx*, *part*)

rdflib.plugins.sparql.evaluate.**evalSelectQuery**(*ctx*, *query*)

rdflib.plugins.sparql.evaluate.**evalSlice**(*ctx*, *slice*)

rdflib.plugins.sparql.evaluate.**evalUnion**(*ctx*, *union*)

rdflib.plugins.sparql.evaluate.**evalValues**(*ctx*, *part*)

evalutils Module

operators Module

This contains evaluation functions for expressions

They get bound as instances-methods to the CompValue objects from parserutils using setEvalFn

rdflib.plugins.sparql.operators.**AdditiveExpression**(*e*, *ctx*)

rdflib.plugins.sparql.operators.**Builtin_ABS**(*expr*, *ctx*)
http://www.w3.org/TR/sparql11-query/#func-abs

rdflib.plugins.sparql.operators.**Builtin_BNODE**(*expr*, *ctx*)
http://www.w3.org/TR/sparql11-query/#func-bnode

rdflib.plugins.sparql.operators.**Builtin_BOUND**(*e*, *ctx*)
http://www.w3.org/TR/sparql11-query/#func-bound

rdflib.plugins.sparql.operators.**Builtin_CEIL**(*expr*, *ctx*)
http://www.w3.org/TR/sparql11-query/#func-ceil

rdflib.plugins.sparql.operators.**Builtin_COALESCE**(*expr*, *ctx*)
http://www.w3.org/TR/sparql11-query/#func-coalesce

rdflib.plugins.sparql.operators.**Builtin_CONCAT**(*expr*, *ctx*)
http://www.w3.org/TR/sparql11-query/#func-concat

rdflib.plugins.sparql.operators.**Builtin_CONTAINS**(*expr*, *ctx*)
http://www.w3.org/TR/sparql11-query/#func-strcontains

rdflib.plugins.sparql.operators.**Builtin_DATATYPE**(*e*, *ctx*)

rdflib.plugins.sparql.operators.**Builtin_DAY**(*e*, *ctx*)

rdflib.plugins.sparql.operators.**Builtin_ENCODE_FOR_URI**(*expr*, *ctx*)

rdflib.plugins.sparql.operators.**Builtin_EXISTS**(*e*, *ctx*)

rdflib.plugins.sparql.operators.**Builtin_FLOOR**(*expr*, *ctx*)
http://www.w3.org/TR/sparql11-query/#func-floor

rdflib.plugins.sparql.operators.**Builtin_HOURS**(*e*, *ctx*)

rdflib.plugins.sparql.operators.**Builtin_IF**(*expr*, *ctx*)
http://www.w3.org/TR/sparql11-query/#func-if

rdflib.plugins.sparql.operators.**Builtin_IRI**(*expr*, *ctx*)
http://www.w3.org/TR/sparql11-query/#func-iri

rdflib.plugins.sparql.operators.**Builtin_LANG**(*e*, *ctx*)
http://www.w3.org/TR/sparql11-query/#func-lang

Returns the language tag of ltrl, if it has one. It returns "" if ltrl has no language tag. Note that the RDF data model does not include literals with an empty language tag.

rdflib.plugins.sparql.operators.**Builtin_LANGMATCHES**(*e*, *ctx*)
http://www.w3.org/TR/sparql11-query/#func-langMatches

rdflib.plugins.sparql.operators.**Builtin_LCASE**(*e*, *ctx*)

rdflib.plugins.sparql.operators.**Builtin_MD5**(*expr*, *ctx*)

rdflib.plugins.sparql.operators.**Builtin_MINUTES**(*e*, *ctx*)

rdflib.plugins.sparql.operators.**Builtin_MONTH**(*e*, *ctx*)

rdflib.plugins.sparql.operators.**Builtin_NOW**(*e*, *ctx*)
http://www.w3.org/TR/sparql11-query/#func-now

rdflib.plugins.sparql.operators.**Builtin_RAND**(*expr*, *ctx*)
http://www.w3.org/TR/sparql11-query/#idp2133952

rdflib.plugins.sparql.operators.**Builtin_REGEX**(*expr*, *ctx*)
http://www.w3.org/TR/sparql11-query/#func-regex Invokes the XPath fn:matches function to match text against a regular expression pattern. The regular expression language is defined in XQuery 1.0 and XPath 2.0 Functions and Operators section 7.6.1 Regular Expression Syntax

rdflib.plugins.sparql.operators.**Builtin_REPLACE**(*expr*, *ctx*)
http://www.w3.org/TR/sparql11-query/#func-substr

rdflib.plugins.sparql.operators.**Builtin_ROUND**(*expr*, *ctx*)
http://www.w3.org/TR/sparql11-query/#func-round

rdflib.plugins.sparql.operators.**Builtin_SECONDS**(*e*, *ctx*)
http://www.w3.org/TR/sparql11-query/#func-seconds

rdflib.plugins.sparql.operators.**Builtin_SHA1**(*expr*, *ctx*)

rdflib.plugins.sparql.operators.**Builtin_SHA256**(*expr*, *ctx*)

rdflib.plugins.sparql.operators.**Builtin_SHA384**(*expr*, *ctx*)

rdflib.plugins.sparql.operators.**Builtin_SHA512**(*expr*, *ctx*)

rdflib.plugins.sparql.operators.**Builtin_STR**(*e*, *ctx*)

rdflib.plugins.sparql.operators.**Builtin_STRAFTER**(*expr*, *ctx*)
http://www.w3.org/TR/sparql11-query/#func-strafter

rdflib.plugins.sparql.operators.**Builtin_STRBEFORE**(*expr*, *ctx*)
http://www.w3.org/TR/sparql11-query/#func-strbefore

rdflib.plugins.sparql.operators.**Builtin_STRDT**(*expr*, *ctx*)
http://www.w3.org/TR/sparql11-query/#func-strdt

rdflib.plugins.sparql.operators.**Builtin_STRENDS**(*expr*, *ctx*)
http://www.w3.org/TR/sparql11-query/#func-strends

rdflib.plugins.sparql.operators.**Builtin_STRLANG**(*expr*, *ctx*)
http://www.w3.org/TR/sparql11-query/#func-strlang

rdflib.plugins.sparql.operators.**Builtin_STRLEN**(*e*, *ctx*)

rdflib.plugins.sparql.operators.**Builtin_STRSTARTS**(*expr*, *ctx*)
http://www.w3.org/TR/sparql11-query/#func-strstarts

rdflib.plugins.sparql.operators.**Builtin_STRUUID**(*expr*, *ctx*)
http://www.w3.org/TR/sparql11-query/#func-strdt

rdflib.plugins.sparql.operators.**Builtin_SUBSTR**(*expr*, *ctx*)
http://www.w3.org/TR/sparql11-query/#func-substr

rdflib.plugins.sparql.operators.**Builtin_TIMEZONE**(*e*, *ctx*)
http://www.w3.org/TR/sparql11-query/#func-timezone

Returns the timezone part of arg as an xsd:dayTimeDuration.

Raises an error if there is no timezone.

rdflib.plugins.sparql.operators.**Builtin_TZ**(*e*, *ctx*)

rdflib.plugins.sparql.operators.**Builtin_UCASE**(*e*, *ctx*)

rdflib.plugins.sparql.operators.**Builtin_UUID**(*expr*, *ctx*)
http://www.w3.org/TR/sparql11-query/#func-strdt

rdflib.plugins.sparql.operators.**Builtin_YEAR**(*e*, *ctx*)

rdflib.plugins.sparql.operators.**Builtin_isBLANK**(*expr*, *ctx*)

rdflib.plugins.sparql.operators.**Builtin_isIRI**(*expr*, *ctx*)

rdflib.plugins.sparql.operators.**Builtin_isLITERAL**(*expr*, *ctx*)

rdflib.plugins.sparql.operators.**Builtin_isNUMERIC**(*expr*, *ctx*)

rdflib.plugins.sparql.operators.**Builtin_sameTerm**(*e*, *ctx*)

rdflib.plugins.sparql.operators.**ConditionalAndExpression**(*e*, *ctx*)

rdflib.plugins.sparql.operators.**ConditionalOrExpression**(*e*, *ctx*)

rdflib.plugins.sparql.operators.**EBV**(*rt*)

- If the argument is a typed literal with a datatype of xsd:boolean, the EBV is the value of that argument.
- If the argument is a plain literal or a typed literal with a datatype of xsd:string, the EBV is false if the operand value has zero length; otherwise the EBV is true.
- If the argument is a numeric type or a typed literal with a datatype derived from a numeric type, the EBV is false if the operand value is NaN or is numerically equal to zero; otherwise the EBV is true.

•All other arguments, including unbound arguments, produce a type error.

rdflib.plugins.sparql.operators.**Function**(*e*, *ctx*)
Custom functions (and casts!)

rdflib.plugins.sparql.operators.**MultiplicativeExpression**(*e*, *ctx*)

rdflib.plugins.sparql.operators.**RelationalExpression**(*e*, *ctx*)

rdflib.plugins.sparql.operators.**UnaryMinus**(*expr*, *ctx*)

rdflib.plugins.sparql.operators.**UnaryNot**(*expr*, *ctx*)

rdflib.plugins.sparql.operators.**UnaryPlus**(*expr*, *ctx*)

rdflib.plugins.sparql.operators.**and_**(**args*)

rdflib.plugins.sparql.operators.**datetime**(*e*)

rdflib.plugins.sparql.operators.**literal**(*s*)

rdflib.plugins.sparql.operators.**not_**(*arg*)

rdflib.plugins.sparql.operators.**numeric**(*expr*)
return a number from a literal http://www.w3.org/TR/xpath20/#promotion

or TypeError

rdflib.plugins.sparql.operators.**simplify**(*expr*)

rdflib.plugins.sparql.operators.**string**(*s*)
Make sure the passed thing is a string literal i.e. plain literal, xsd:string literal or lang-tagged literal

parser Module SPARQL 1.1 Parser

based on pyparsing

rdflib.plugins.sparql.parser.**expandBNodeTriples**(*terms*)
expand [?p ?o] syntax for implicit bnodes

rdflib.plugins.sparql.parser.**expandCollection**(*terms*)
expand (1 2 3) notation for collections

rdflib.plugins.sparql.parser.**expandTriples**(*terms*)
Expand ; and , syntax for repeat predicates, subjects

rdflib.plugins.sparql.parser.**expandUnicodeEscapes**(*q*)
The syntax of the SPARQL Query Language is expressed over code points in Unicode [UNICODE]. The encoding is always UTF-8 [RFC3629]. Unicode code points may also be expressed using an uXXXX (U+0 to U+FFFF) or UXXXXXXXX syntax (for U+10000 onwards) where X is a hexadecimal digit [0-9A-F]

rdflib.plugins.sparql.parser.**neg**(*literal*)

rdflib.plugins.sparql.parser.**parseQuery**(*q*)

rdflib.plugins.sparql.parser.**parseUpdate**(*q*)

rdflib.plugins.sparql.parser.**setDataType**(*terms*)

rdflib.plugins.sparql.parser.**setLanguage**(*terms*)

parserutils Module

class rdflib.plugins.sparql.parserutils.**Comp**(*name*, *expr*)
Bases: pyparsing.TokenConverter

A pyparsing token for grouping together things with a label Any sub-tokens that are not Params will be ignored.

Returns CompValue / Expr objects - depending on whether evalFn is set.

__init__(*name*, *expr*)

__module__ = 'rdflib.plugins.sparql.parserutils'

__slotnames__ = []

postParse(*instring*, *loc*, *tokenList*)

setEvalFn(*evalfn*)

class rdflib.plugins.sparql.parserutils.**CompValue**(*name*, ***values*)
Bases: collections.OrderedDict

The result of parsing a Comp Any included Params are avaiable as Dict keys or as attributes

__getattr__(*a*)

__getitem__(*a*)

__init__(*name*, ***values*)

__module__ = 'rdflib.plugins.sparql.parserutils'

__repr__()

__str__()

get(*a*, *variables=False*, *errors=False*)

class rdflib.plugins.sparql.parserutils.**Expr**(*name*, *evalfn=None*, ***values*)
Bases: *rdflib.plugins.sparql.parserutils.CompValue*

A CompValue that is evaluatable

__init__(*name*, *evalfn=None*, ***values*)

__module__ = 'rdflib.plugins.sparql.parserutils'

eval(*ctx={}*)

class rdflib.plugins.sparql.parserutils.**Param**(*name*, *expr*, *isList=False*)
Bases: pyparsing.TokenConverter

A pyparsing token for labelling a part of the parse-tree if isList is true repeat occurrences of ParamList have their values merged in a list

__init__(*name*, *expr*, *isList=False*)

__module__ = 'rdflib.plugins.sparql.parserutils'

__slotnames__ = []

postParse2(*tokenList*)

class rdflib.plugins.sparql.parserutils.**ParamList**(*name*, *expr*)
Bases: *rdflib.plugins.sparql.parserutils.Param*

A shortcut for a Param with isList=True

__init__(*name*, *expr*)

__module__ = 'rdflib.plugins.sparql.parserutils'

class rdflib.plugins.sparql.parserutils.**ParamValue**(*name*, *tokenList*, *isList*)
Bases: object

The result of parsing a Param This just keeps the name/value All cleverness is in the CompValue

__init__(*name*, *tokenList*, *isList*)

__module__ = 'rdflib.plugins.sparql.parserutils'

__str__()

class rdflib.plugins.sparql.parserutils.**plist**
Bases: list

this is just a list, but we want our own type to check for

__module__ = 'rdflib.plugins.sparql.parserutils'

rdflib.plugins.sparql.parserutils.**value**(*ctx*, *val*, *variables=False*, *errors=False*)
utility function for evaluating something...

Variables will be looked up in the context Normally, non-bound vars is an error, set variables=True to return unbound vars

Normally, an error raises the error, set errors=True to return error

processor Module Code for tying SPARQL Engine into RDFLib

These should be automatically registered with RDFLib

class rdflib.plugins.sparql.processor.**SPARQLProcessor**(*graph*)
Bases: *rdflib.query.Processor*

__init__(*graph*)

__module__ = 'rdflib.plugins.sparql.processor'

query(*strOrQuery*, *initBindings={}*, *initNs={}*, *base=None*, *DEBUG=False*)
Evaluate a query with the given initial bindings, and initial namespaces. The given base is used to resolve relative URIs in the query and will be overridden by any BASE given in the query.

class rdflib.plugins.sparql.processor.**SPARQLResult**(*res*)
Bases: *rdflib.query.Result*

__init__(*res*)

__module__ = 'rdflib.plugins.sparql.processor'

class rdflib.plugins.sparql.processor.**SPARQLUpdateProcessor**(*graph*)
Bases: rdflib.query.UpdateProcessor

__init__(*graph*)

__module__ = 'rdflib.plugins.sparql.processor'

update(*strOrQuery*, *initBindings={}*, *initNs={}*)

rdflib.plugins.sparql.processor.**prepareQuery**(*queryString*, *initNs={}*, *base=None*)
Parse and translate a SPARQL Query

rdflib.plugins.sparql.processor.**processUpdate**(*graph*, *updateString*, *initBindings={}*, *initNs={}*, *base=None*)
Process a SPARQL Update Request returns Nothing on success or raises Exceptions on error

sparql Module

exception rdflib.plugins.sparql.sparql.**AlreadyBound**

Bases: *rdflib.plugins.sparql.sparql.SPARQLError*

Raised when trying to bind a variable that is already bound!

__init__()

__module__ = 'rdflib.plugins.sparql.sparql'

class rdflib.plugins.sparql.sparql.**Bindings**(*outer=None*, *d=[]*)

Bases: _abcoll.MutableMapping

A single level of a stack of variable-value bindings. Each dict keeps a reference to the dict below it, any failed lookup is propegated back

In python 3.3 this could be a collections.ChainMap

__abstractmethods__ = frozenset([])

__contains__(*key*)

__delitem__(*key*)

__getitem__(*key*)

__init__(*outer=None*, *d=[]*)

__iter__()

__len__()

__module__ = 'rdflib.plugins.sparql.sparql'

__repr__()

__setitem__(*key*, *value*)

__str__()

class rdflib.plugins.sparql.sparql.**FrozenBindings**(*ctx*, **args*, ***kwargs*)

Bases: *rdflib.plugins.sparql.sparql.FrozenDict*

__abstractmethods__ = frozenset([])

__getitem__(*key*)

__init__(*ctx*, **args*, ***kwargs*)

__module__ = 'rdflib.plugins.sparql.sparql'

bnodes

forget(*before*)

return a frozen dict only of bindings made in self since before

merge(*other*)

now

project(*vars*)

prologue

remember(*these*)

return a frozen dict only of bindings in these

class rdflib.plugins.sparql.sparql.**FrozenDict**(**args*, ***kwargs*)
Bases: _abcoll.Mapping

An immutable hashable dict

Taken from http://stackoverflow.com/a/2704866/81121

__abstractmethods__ = frozenset([])

__getitem__(*key*)

__hash__()

__init__(**args*, ***kwargs*)

__iter__()

__len__()

__module__ = 'rdflib.plugins.sparql.sparql'

__repr__()

__str__()

compatible(*other*)

disjointDomain(*other*)

merge(*other*)

project(*vars*)

exception rdflib.plugins.sparql.sparql.**NotBoundError**(*msg=None*)
Bases: *rdflib.plugins.sparql.sparql.SPARQLError*

__init__(*msg=None*)

__module__ = 'rdflib.plugins.sparql.sparql'

class rdflib.plugins.sparql.sparql.**Prologue**
A class for holding prefixing bindings and base URI information

__init__()

__module__ = 'rdflib.plugins.sparql.sparql'

absolutize(*iri*)
Apply BASE / PREFIXes to URIs (and to datatypes in Literals)

TODO: Move resolving URIs to pre-processing

bind(*prefix*, *uri*)

resolvePName(*prefix*, *localname*)

class rdflib.plugins.sparql.sparql.**Query**(*prologue*, *algebra*)
A parsed and translated query

__init__(*prologue*, *algebra*)

__module__ = 'rdflib.plugins.sparql.sparql'

class rdflib.plugins.sparql.sparql.**QueryContext**(*graph=None*, *bindings=None*)
Bases: object

Query context - passed along when evaluating the query

__getitem__(*key*)

__init__(*graph=None*, *bindings=None*)

__module__ = 'rdflib.plugins.sparql.sparql'

__setitem__(*key*, *value*)

clean()

clone(*bindings=None*)

dataset
current dataset

get(*key*, *default=None*)

load(*source*, *default=False*, ***kwargs*)

push()

pushGraph(*graph*)

solution(*vars=None*)
Return a static copy of the current variable bindings as dict

thaw(*frozenbindings*)
Create a new read/write query context from the given solution

exception rdflib.plugins.sparql.sparql.**SPARQLError**(*msg=None*)
Bases: exceptions.Exception

__init__(*msg=None*)

__module__ = 'rdflib.plugins.sparql.sparql'

exception rdflib.plugins.sparql.sparql.**SPARQLTypeError**(*msg*)
Bases: *rdflib.plugins.sparql.sparql.SPARQLError*

__init__(*msg*)

__module__ = 'rdflib.plugins.sparql.sparql'

update Module Code for carrying out Update Operations

rdflib.plugins.sparql.update.**evalAdd**(*ctx*, *u*)
add all triples from src to dst

http://www.w3.org/TR/sparql11-update/#add

rdflib.plugins.sparql.update.**evalClear**(*ctx*, *u*)
http://www.w3.org/TR/sparql11-update/#clear

rdflib.plugins.sparql.update.**evalCopy**(*ctx*, *u*)
remove all triples from dst add all triples from src to dst

http://www.w3.org/TR/sparql11-update/#copy

rdflib.plugins.sparql.update.**evalCreate**(*ctx*, *u*)
http://www.w3.org/TR/sparql11-update/#create

rdflib.plugins.sparql.update.**evalDeleteData**(*ctx*, *u*)
http://www.w3.org/TR/sparql11-update/#deleteData

rdflib.plugins.sparql.update.**evalDeleteWhere**(*ctx*, *u*)
http://www.w3.org/TR/sparql11-update/#deleteWhere

rdflib.plugins.sparql.update.**evalDrop**(*ctx*, *u*)
http://www.w3.org/TR/sparql11-update/#drop

rdflib.plugins.sparql.update.**evalInsertData**(*ctx*, *u*)
http://www.w3.org/TR/sparql11-update/#insertData

rdflib.plugins.sparql.update.**evalLoad**(*ctx*, *u*)
http://www.w3.org/TR/sparql11-update/#load

rdflib.plugins.sparql.update.**evalModify**(*ctx*, *u*)

rdflib.plugins.sparql.update.**evalMove**(*ctx*, *u*)
remove all triples from dst add all triples from src to dst remove all triples from src

http://www.w3.org/TR/sparql11-update/#move

rdflib.plugins.sparql.update.**evalUpdate**(*graph*, *update*, *initBindings=None*)
http://www.w3.org/TR/sparql11-update/#updateLanguage

'A request is a sequence of operations [...] Implementations MUST ensure that operations of a single request are executed in a fashion that guarantees the same effects as executing them in lexical order.

Operations all result either in success or failure.

If multiple operations are present in a single request, then a result of failure from any operation MUST abort the sequence of operations, causing the subsequent operations to be ignored.'

This will return None on success and raise Exceptions on error

Subpackages

results Package

csvresults Module This module implements a parser and serializer for the CSV SPARQL result formats

http://www.w3.org/TR/sparql11-results-csv-tsv/

class rdflib.plugins.sparql.results.csvresults.**CSVResultParser**
Bases: *rdflib.query.ResultParser*

__init__()

__module__ = 'rdflib.plugins.sparql.results.csvresults'

convertTerm(*t*)

parse(*source*)

parseRow(*row*, *v*)

class rdflib.plugins.sparql.results.csvresults.**CSVResultSerializer**(*result*)
Bases: *rdflib.query.ResultSerializer*

__init__(*result*)

__module__ = 'rdflib.plugins.sparql.results.csvresults'

serialize(*stream*, *encoding='utf-8'*)

serializeTerm(*term*, *encoding*)

jsonlayer Module Thin abstraction layer over the different available modules for decoding and encoding JSON data.

This module currently supports the following JSON modules:

- `simplejson`: http://code.google.com/p/simplejson/
- `cjson`: http://pypi.python.org/pypi/python-cjson
- `json`: This is the version of `simplejson` that is bundled with the Python standard library since version 2.6 (see http://docs.python.org/library/json.html)

The default behavior is to use `simplejson` if installed, and otherwise fallback to the standard library module. To explicitly tell SPARQLWrapper which module to use, invoke the *use()* function with the module name:

```
import jsonlayer
jsonlayer.use('cjson')
```

In addition to choosing one of the above modules, you can also configure SPARQLWrapper to use custom decoding and encoding functions:

```
import jsonlayer
jsonlayer.use(decode=my_decode, encode=my_encode)
```

rdflib.plugins.sparql.results.jsonlayer.**decode**(*string*)
Decode the given JSON string.

Parameters **string** (*basestring*) – the JSON string to decode

Returns the corresponding Python data structure

Return type *object*

rdflib.plugins.sparql.results.jsonlayer.**encode**(*obj*)
Encode the given object as a JSON string.

Parameters **obj** (`object`) – the Python data structure to encode

Returns the corresponding JSON string

Return type basestring

rdflib.plugins.sparql.results.jsonlayer.**use**(*module=None*, *decode=None*, *encode=None*)
Set the JSON library that should be used, either by specifying a known module name, or by providing a decode and encode function.

The modules "simplejson", "cjson", and "json" are currently supported for the `module` parameter.

If provided, the `decode` parameter must be a callable that accepts a JSON string and returns a corresponding Python data structure. The `encode` callable must accept a Python data structure and return the corresponding JSON string. Exceptions raised by decoding and encoding should be propagated up unaltered.

Parameters

- **module** (*str or module*) – the name of the JSON library module to use, or the module object itself
- **decode** (*callable*) – a function for decoding JSON strings
- **encode** (*callable*) – a function for encoding objects as JSON strings

jsonresults Module

class rdflib.plugins.sparql.results.jsonresults.**JSONResult**(*json*)
Bases: *rdflib.query.Result*

__init__(*json*)

__module__ = 'rdflib.plugins.sparql.results.jsonresults'

class rdflib.plugins.sparql.results.jsonresults.**JSONResultParser**
Bases: *rdflib.query.ResultParser*

__module__ = 'rdflib.plugins.sparql.results.jsonresults'

parse(*source*)

class rdflib.plugins.sparql.results.jsonresults.**JSONResultSerializer**(*result*)
Bases: *rdflib.query.ResultSerializer*

__init__(*result*)

__module__ = 'rdflib.plugins.sparql.results.jsonresults'

serialize(*stream*, *encoding=None*)

rdflib.plugins.sparql.results.jsonresults.**parseJsonTerm**(*d*)
rdflib object (Literal, URIRef, BNode) for the given json-format dict.

input is like: { 'type': 'uri', 'value': 'http://famegame.com/2006/01/username' } { 'type': 'literal', 'value': 'drewp' }

rdflib.plugins.sparql.results.jsonresults.**termToJSON**(*self*, *term*)

rdfresults Module

class rdflib.plugins.sparql.results.rdfresults.**RDFResult**(*source*, ***kwargs*)
Bases: *rdflib.query.Result*

__init__(*source*, ***kwargs*)

__module__ = 'rdflib.plugins.sparql.results.rdfresults'

class rdflib.plugins.sparql.results.rdfresults.**RDFResultParser**
Bases: *rdflib.query.ResultParser*

__module__ = 'rdflib.plugins.sparql.results.rdfresults'

parse(*source*, ***kwargs*)

tsvresults Module This implements the Tab Separated SPARQL Result Format

It is implemented with pyparsing, reusing the elements from the SPARQL Parser

class rdflib.plugins.sparql.results.tsvresults.**TSVResultParser**
Bases: *rdflib.query.ResultParser*

__module__ = 'rdflib.plugins.sparql.results.tsvresults'

convertTerm(*t*)

parse(*source*)

xmlresults Module

rdflib.plugins.sparql.results.xmlresults.**RESULTS_NS_ET = u'{http://www.w3.org/2005/sparql-results#}'**

A Parser for SPARQL results in XML:

http://www.w3.org/TR/rdf-sparql-XMLres/

Bits and pieces borrowed from: http://projects.bigasterisk.com/sparqlhttp/

Authors: Drew Perttula, Gunnar Aastrand Grimnes

class rdflib.plugins.sparql.results.xmlresults.**SPARQLXMLWriter**(*output*, *encoding='utf-8'*)

Python saxutils-based SPARQL XML Writer

__init__(*output*, *encoding='utf-8'*)

__module__ = 'rdflib.plugins.sparql.results.xmlresults'

close()

write_ask(*val*)

write_binding(*name*, *val*)

write_end_result()

write_header(*allvarsL*)

write_results_header()

write_start_result()

class rdflib.plugins.sparql.results.xmlresults.**XMLResult**(*source*)

Bases: *rdflib.query.Result*

__init__(*source*)

__module__ = 'rdflib.plugins.sparql.results.xmlresults'

class rdflib.plugins.sparql.results.xmlresults.**XMLResultParser**

Bases: *rdflib.query.ResultParser*

__module__ = 'rdflib.plugins.sparql.results.xmlresults'

parse(*source*)

class rdflib.plugins.sparql.results.xmlresults.**XMLResultSerializer**(*result*)

Bases: *rdflib.query.ResultSerializer*

__init__(*result*)

__module__ = 'rdflib.plugins.sparql.results.xmlresults'

serialize(*stream*, *encoding='utf-8'*)

rdflib.plugins.sparql.results.xmlresults.**parseTerm**(*element*)

rdflib object (Literal, URIRef, BNode) for the given elementtree element

stores Package

stores Package

This package contains modules for additional RDFLib stores

auditable Module This wrapper intercepts calls through the store interface and implements thread-safe logging of destructive operations (adds / removes) in reverse. This is persisted on the store instance and the reverse operations are executed In order to return the store to the state it was when the transaction began Since the reverse operations are persisted on the store, the store itself acts as a transaction.

Calls to commit or rollback, flush the list of reverse operations This provides thread-safe atomicity and isolation (assuming concurrent operations occur with different store instances), but no durability (transactions are persisted in memory and wont be available to reverse operations after the system fails): A and I out of ACID.

class rdflib.plugins.stores.auditable.**AuditableStore**(*store*)
Bases: *rdflib.store.Store*

__init__(*store*)

__len__(*context=None*)

__module__ = 'rdflib.plugins.stores.auditable'

add(*triple*, *context*, *quoted=False*)

bind(*prefix*, *namespace*)

close(*commit_pending_transaction=False*)

commit()

contexts(*triple=None*)

destroy(*configuration*)

namespace(*prefix*)

namespaces()

open(*configuration*, *create=True*)

prefix(*namespace*)

query(**args*, ***kw*)

remove(*(subject*, *predicate*, *object_)*, *context=None*)

rollback()

triples(*triple*, *context=None*)

concurrent Module

class rdflib.plugins.stores.concurrent.**ConcurrentStore**(*store*)
Bases: object

__init__(*store*)

__len__()

__module__ = 'rdflib.plugins.stores.concurrent'

add(*triple*)

remove(*triple*)

triples(*triple*)

class rdflib.plugins.stores.concurrent.**ResponsibleGenerator**(*gen*, *cleanup*)
Bases: object

A generator that will help clean up when it is done being used.

__del__()

__init__(*gen*, *cleanup*)

__iter__()

__module__ = 'rdflib.plugins.stores.concurrent'

__slots__ = ['cleanup', 'gen']

cleanup

gen

next()

regexmatching Module This wrapper intercepts calls through the store interface which make use of the REGEXTerm class to represent matches by REGEX instead of literal comparison.

Implemented for stores that don't support this and essentially provides the support by replacing the REGEXTerms by wildcards (None) and matching against the results from the store it's wrapping.

class rdflib.plugins.stores.regexmatching.**REGEXMatching**(*storage*)
Bases: *rdflib.store.Store*

__init__(*storage*)

__len__(*context=None*)

__module__ = 'rdflib.plugins.stores.regexmatching'

add(*triple*, *context*, *quoted=False*)

bind(*prefix*, *namespace*)

close(*commit_pending_transaction=False*)

commit()

contexts(*triple=None*)

destroy(*configuration*)

namespace(*prefix*)

namespaces()

open(*configuration*, *create=True*)

prefix(*namespace*)

remove(*triple*, *context=None*)

remove_context(*identifier*)

rollback()

triples(*triple*, *context=None*)

class rdflib.plugins.stores.regexmatching.**REGEXTerm**(*expr*)
Bases: unicode

REGEXTerm can be used in any term slot and is interpreted as a request to perform a REGEX match (not a string comparison) using the value (pre-compiled) for checking rdf:type matches

__init__(*expr*)

__module__ = 'rdflib.plugins.stores.regexmatching'

__reduce__()

rdflib.plugins.stores.regexmatching.**regexCompareQuad**(*quad*, *regexQuad*)

sparqlstore Module This is an RDFLib store around Ivan Herman et al.'s SPARQL service wrapper. This was first done in layer-cake, and then ported to RDFLib

rdflib.plugins.stores.sparqlstore.**CastToTerm**(*node*)
Helper function that casts XML node in SPARQL results to appropriate rdflib term

class rdflib.plugins.stores.sparqlstore.**NSSPARQLWrapper**(*endpoint*, *updateEndpoint=None*, *returnFormat='xml'*, *defaultGraph=None*, *agent='sparqlwrapper 1.8.0 (rdflib.github.io/sparqlwrapper)'*)

Bases: SPARQLWrapper.Wrapper.SPARQLWrapper

__module__ = 'rdflib.plugins.stores.sparqlstore'

injectPrefixes(*query*)

nsBindings = {}

setNamespaceBindings(*bindings*)
A shortcut for setting namespace bindings that will be added to the prolog of the query

@param bindings: A dictionary of prefixs to URIs

setQuery(*query*)
Set the SPARQL query text. Note: no check is done on the validity of the query (syntax or otherwise) by this module, except for testing the query type (SELECT, ASK, etc).

Syntax and validity checking is done by the SPARQL service itself.

@param query: query text @type query: string @bug: #2320024

class rdflib.plugins.stores.sparqlstore.**SPARQLStore**(*endpoint=None*, *bNodeAsURI=False*, *sparql11=True*, *context_aware=True*, ***sparqlwrapper_kwargs*)

Bases: *rdflib.plugins.stores.sparqlstore.NSSPARQLWrapper*, *rdflib.store.Store*

An RDFLib store around a SPARQL endpoint

This is in theory context-aware and should work as expected when a context is specified.

For ConjunctiveGraphs, reading is done from the "default graph". Exactly what this means depends on your endpoint, because SPARQL does not offer a simple way to query the union of all graphs as it would be expected for a ConjuntiveGraph. This is why we recommend using Dataset instead, which is motivated by the SPARQL 1.1.

Fuseki/TDB has a flag for specifying that the default graph is the union of all graphs (tdb:unionDefaultGraph in the Fuseki config).

> **Warning:** The SPARQL Store does not support blank-nodes!
> As blank-nodes act as variables in SPARQL queries there is no way to query for a particular blank node.
> See http://www.w3.org/TR/sparql11-query/#BGPsparqlBNodes

__init__(*endpoint=None*, *bNodeAsURI=False*, *sparql11=True*, *context_aware=True*, ***sparqlwrapper_kwargs*)

__len__(*context=None*)

__module__ = 'rdflib.plugins.stores.sparqlstore'

add(*(subject, predicate, obj), context=None, quoted=False*)

addN(*quads*)

add_graph(*graph*)

bind(*prefix, namespace*)

commit()

contexts(*triple=None*)

Iterates over results to "SELECT ?NAME { GRAPH ?NAME { ?s ?p ?o } }" or "SELECT ?NAME { GRAPH ?NAME {} }" if triple is *None*.

Returns instances of this store with the SPARQL wrapper object updated via addNamedGraph(?NAME).

This causes a named-graph-uri key / value pair to be sent over the protocol.

Please note that some SPARQL endpoints are not able to find empty named graphs.

create(*configuration*)

destroy(*configuration*)

formula_aware = False

graph_aware = True

namespace(*prefix*)

namespaces()

open(*configuration, create=False*)

sets the endpoint URL for this SPARQLStore if create==True an exception is thrown.

prefix(*namespace*)

query(*query, initNs={}, initBindings={}, queryGraph=None, DEBUG=False*)

query_endpoint

regex_matching = 0

remove(*(subject, predicate, obj), context*)

remove_graph(*graph*)

rollback()

transaction_aware = False

triples(*(s, p, o), context=None*)

- **tuple (s, o, p)** the triple used as filter for the SPARQL select. (None, None, None) means anything.
- **context context** the graph effectively calling this method.

Returns a tuple of triples executing essentially a SPARQL like SELECT ?subj ?pred ?obj WHERE { ?subj ?pred ?obj }

context may include three parameter to refine the underlying query:

- LIMIT: an integer to limit the number of results
- OFFSET: an integer to enable paging of results
- ORDERBY: an instance of Variable('s'), Variable('o') or Variable('p')

or, by default, the first 'None' from the given triple

•Using LIMIT or OFFSET automatically include ORDERBY otherwise this is

because the results are retrieved in a not deterministic way (depends on the walking path on the graph) - Using OFFSET without defining LIMIT will discard the first OFFSET - 1 results

'' a_graph.LIMIT = limit a_graph.OFFSET = offset triple_generator = a_graph.triples(mytriple):

#do something

#Removes LIMIT and OFFSET if not required for the next triple() calls del a_graph.LIMIT del a_graph.OFFSET ''

triples_choices(*(subject*, *predicate*, *object_)*, *context=None*)

A variant of triples that can take a list of terms instead of a single term in any slot. Stores can implement this to optimize the response time from the import default 'fallback' implementation, which will iterate over each term in the list and dispatch to triples.

class rdflib.plugins.stores.sparqlstore.**SPARQLUpdateStore**(*queryEndpoint=None*, *update_endpoint=None*, *bNodeAsURI=False*, *sparql11=True*, *context_aware=True*, *postAsEncoded=True*, *autocommit=True*)

Bases: *rdflib.plugins.stores.sparqlstore.SPARQLStore*

A store using SPARQL queries for reading and SPARQL Update for changes.

This can be context-aware, if so, any changes will be to the given named graph only.

In favor of the SPARQL 1.1 motivated Dataset, we advise against using this with ConjunctiveGraphs, as it reads and writes from and to the "default graph". Exactly what this means depends on the endpoint and can result in confusion.

For Graph objects, everything works as expected.

> **Warning:** The SPARQL Update Store does not support blank-nodes!
> As blank-nodes acts as variables in SPARQL queries there is no way to query for a particular blank node.
> See http://www.w3.org/TR/sparql11-query/#BGPsparqlBNodes

BLOCK_END = u'}'

BLOCK_FINDING_PATTERN = <_sre.SRE_Pattern object at 0x38ff000>

BLOCK_START = u'{'

BlockContent = u'((\'([^\'\\\\]|\\\\.)*\')|("([^"\\\\]|\\\\.)*")|(\'\'\'((\'|\'\')?([^\'\\\\]|\\\\.))*\'\'\')|("""(("|"")?([^"\\\\]|\\\\.))*""")

BlockFinding = u'(?P<block_start>{)|(?P<block_end>})|(?P<block_content>((\'([^\'\\\\]|\\\\.)*\')|("([^"\\\\]|\\\\.)*")|(\'\'\'

COMMENT = u'#[^\\x0D\\x0A]*([\\x0D\\x0A]|\\Z)'

ESCAPED = u'\\\\.'

IRIREF = u'<([^<>"{}|^`\\]\\\\\\[\\x00-\\x20])*>'

STRING_LITERAL1 = u"'([^'\\\\]|\\\\.)*'"

STRING_LITERAL2 = u'"([^"\\\\]|\\\\.)*"'

STRING_LITERAL_LONG1 = u"''('('|'')?([^'\\\\]|\\\\.))*''"

STRING_LITERAL_LONG2 = u'""("("|"")?([^"\\\\]|\\\\.))*""'

String = u'(\'([^\'\\\\]|\\\\.)*\')|("([^"\\\\]|\\\\.)*")|(\'\'\'((\'|\'\')?([^\'\\\\]|\\\\.))*\'\'\')|("""(("|"")?([^"\\\\]|\\\\.))*""")'

__init__(*queryEndpoint=None*, *update_endpoint=None*, *bNodeAsURI=False*, *sparql11=True*, *context_aware=True*, *postAsEncoded=True*, *autocommit=True*)

__len__(**args*, ***kwargs*)

__module__ = 'rdflib.plugins.stores.sparqlstore'

add(*spo*, *context=None*, *quoted=False*)
Add a triple to the store of triples.

addN(*quads*)
Add a list of quads to the store.

add_graph(*graph*)

commit()
add(), addN(), and remove() are transactional to reduce overhead of many small edits. Read and update() calls will automatically commit any outstanding edits. This should behave as expected most of the time, except that alternating writes and reads can degenerate to the original call-per-triple situation that originally existed.

contexts(**args*, ***kwargs*)

open(*configuration*, *create=False*)
sets the endpoint URLs for this SPARQLStore :param configuration: either a tuple of (queryEndpoint, update_endpoint),

or a string with the query endpoint

Parameters **create** – if True an exception is thrown.

query(**args*, ***kwargs*)

remove(*spo*, *context*)
Remove a triple from the store

remove_graph(*graph*)

rollback()

triples(**args*, ***kwargs*)

update(*query*, *initNs={}*, *initBindings={}*, *queryGraph=None*, *DEBUG=False*)
Perform a SPARQL Update Query against the endpoint, INSERT, LOAD, DELETE etc. Setting initNs adds PREFIX declarations to the beginning of the update. Setting initBindings adds inline VALUEs to the beginning of every WHERE clause. By the SPARQL grammar, all operations that support variables (namely INSERT and DELETE) require a WHERE clause. Important: initBindings fails if the update contains the substring 'WHERE {' which does not denote a WHERE clause, e.g. if it is part of a literal.

Context-aware query rewriting

- **When:** If context-awareness is enabled and the graph is not the default graph of the store.
- **Why:** To ensure consistency with the *IOMemory* store. The graph must except "local" SPARQL requests (requests with no GRAPH keyword) like if it was the default graph.
- **What is done:** These "local" queries are rewritten by this store. The content of each block of a SPARQL Update operation is wrapped in a GRAPH block except if the block is empty. This basically causes INSERT, INSERT DATA, DELETE, DELETE DATA and WHERE to operate only on the context.

- **Example:** *"INSERT DATA { <urn:michel> <urn:likes> <urn:pizza> }"* is converted into *"INSERT DATA { GRAPH <urn:graph> { <urn:michel> <urn:likes> <urn:pizza> } }"*.
- **Warning:** Queries are presumed to be "local" but this assumption is **not checked**. For instance, if the query already contains GRAPH blocks, the latter will be wrapped in new GRAPH blocks.
- **Warning:** A simplified grammar is used that should tolerate extensions of the SPARQL grammar. Still, the process may fail in uncommon situations and produce invalid output.

update_endpoint
the HTTP URL for the Update endpoint, typicallysomething like http://server/dataset/update

where_pattern = <_sre.SRE_Pattern object>

rdflib.plugins.stores.sparqlstore.**TraverseSPARQLResultDOM**(*doc*, *asDictionary=False*)

Returns a generator over tuples of results

rdflib.plugins.stores.sparqlstore.**localName**(*qname*)

tools Package

These commandline-tools are installed into *INSTALL_PREFIX*/bin by setuptools.

tools Package Various commandline tools for working with RDFLib

csv2rdf Module A commandline tool for semi-automatically converting CSV to RDF

try: csv2rdf --help

class rdflib.tools.csv2rdf.**CSV2RDF**
Bases: object

__init__()

__module__ = 'rdflib.tools.csv2rdf'

convert(*csvreader*)

triple(*s*, *p*, *o*)

graphisomorphism Module A commandline tool for testing if RDF graphs are isomorpic, i.e. equal if BNode labels are ignored.

class rdflib.tools.graphisomorphism.**IsomorphicTestableGraph**(**kargs*)
Bases: *rdflib.graph.Graph*

Ported from: http://www.w3.org/2001/sw/DataAccess/proto-tests/tools/rdfdiff.py (Sean B Palmer's RDF Graph Isomorphism Tester)

__eq__(*G*)
Graph isomorphism testing.

__init__(**kargs*)

__module__ = 'rdflib.tools.graphisomorphism'

__ne__(*G*)
Negative graph isomorphism testing.

hashtriples()

internal_hash()
This is defined instead of __hash__ to avoid a circular recursion scenario with the Memory store for rdflib which requires a hash lookup in order to return a generator of triples

vhash(*term*, *done=False*)

vhashtriple(*triple*, *term*, *done*)

vhashtriples(*term*, *done*)

rdflib.tools.graphisomorphism.**main**()

rdf2dot Module A commandline tool for drawing RDF graphs in Graphviz DOT format

You can draw the graph of an RDF file directly:

rdflib.tools.rdf2dot.**main**()

rdflib.tools.rdf2dot.**rdf2dot**(*g*, *stream*, *opts={}*)
Convert the RDF graph to DOT writes the dot output to the stream

rdfpipe Module A commandline tool for parsing RDF in different formats and serializing the resulting graph to a chosen format.

rdflib.tools.rdfpipe.**main**()

rdflib.tools.rdfpipe.**make_option_parser**()

rdflib.tools.rdfpipe.**parse_and_serialize**(*input_files*, *input_format*, *guess*, *outfile*, *output_format*, *ns_bindings*, *store_conn=''*, *store_type=None*)

rdfs2dot Module A commandline tool for drawing RDFS Class diagrams in Graphviz DOT format

You can draw the graph of an RDFS file directly:

rdflib.tools.rdfs2dot.**main**()

rdflib.tools.rdfs2dot.**rdfs2dot**(*g*, *stream*, *opts={}*)
Convert the RDFS schema in a graph writes the dot output to the stream

- genindex
- modindex

CHAPTER 4

For developers

RDFLib developers guide

Introduction

This document describes the process and conventions to follow when developing RDFLib code.

Please be as Pythonic as possible (**PEP 8**).

Code will occasionally be auto-formatted using `autopep8` - you can also do this yourself.

Any new functionality being added to RDFLib should have doc tests and unit tests. Tests should be added for any functionality being changed that currently does not have any doc tests or unit tests. And all the tests should be run before committing changes to make sure the changes did not break anything.

If you add a new cool feature, consider also adding an example in `./examples`

Running tests

Run tests with nose:

Specific tests can either be run by module name or file name. For example:

```
$ python run_tests.py --tests rdflib.graph
$ python run_tests.py --tests test/test_graph.py
```

Writing documentation

We use sphinx for generating HTML docs, see *Writing RDFLib Documentation*

Continous Integration

We used Travis for CI, see:

https://travis-ci.org/RDFLib/rdflib

If you make a pull-request to RDFLib on GitHub, travis will automatically test you code.

Compatibility

RDFLib>=3.X tries to be compatible with python versions 2.5 - 3

Some of the limitations we've come across:

- Python 2.5/2.6 has no abstract base classes from collections, such `MutableMap`, etc.
- 2.5/2.6 No skipping tests using `unittest`, i.e. `TestCase.skipTest` and decorators are missing => use nose instead
- no `str.decode('string-escape')` in py3
- no `json` module in 2.5 (install `simplejson` instead)
- no `ordereddict` in 2.5/2.6 (install `ordereddict` module)
- `collections.Counter` was added in 2.6

Releasing

Set to-be-released version number in `rdflib/__init__.py` and `README.md`. Check date in `LICENSE`.

Add `CHANGELOG.md` entry.

Commit this change. It's preferable make the release tag via https://github.com/RDFLib/rdflib/releases/new :: Our Tag versions aren't started with 'v', so just use a plain 4.2.0 like version. Release title is like "RDFLib 4.2.0", the description a copy of your `CHANGELOG.md` entry. This gives us a nice release page like this:: https://github.com/RDFLib/rdflib/releases/tag/4.2.0

If for whatever reason you don't want to take this approach, the old one is:

```
Tagging the release commit with::

  git tag -a -m 'tagged version' X.X.X

When pushing, remember to do::

  git push --tags
```

No matter how you create the release tag, remember to upload tarball to pypi with:

```
python setup.py sdist upload
```

Set new dev version number in the above locations, i.e. next release *-dev*: `2.4.1-dev` and commit again.

Update the topic of #rdflib on freenode irc:

```
/msg ChanServ topic #rdflib https://github.com/RDFLib/rdflib | latest stable version: 4.2.0 | docs: h
```

Writing RDFLib Documentation

The docs are generated with Sphinx.

Sphinx makes it very easy to pull in doc-strings from modules, classes, methods, etc. When writing doc-strings, special reST fields can be used to annotate parameters, return-types, etc. This make for pretty API docs:

http://sphinx-doc.org/domains.html?highlight=param#info-field-lists

Building

To build you must have the *sphinx* package installed:

```
pip install sphinx
```

Then you can do:

```
python setup.py build_sphinx
```

The docs will be generated in `build/sphinx/html/`

Syntax highlighting

To get N3 and SPARQL syntax highlighting do:

```
pip install -e git+git://github.com/gjhiggins/sparql_pygments_lexer.git#egg=SPARQL_Pygments_Lexer
pip install -e git+git://github.com/gjhiggins/n3_pygments_lexer.git#egg=Notation3_Pygments_Lexer
```

API Docs

API Docs are automatically generated with `sphinx-apidoc`:

```
sphinx-apidoc -f -d 10 -o docs/apidocs/ rdflib examples
```

(then `rdflib.rst` was tweaked manually to not include all convenience imports that are directly in the `rdflib/__init__.py`)

Tables

The tables in `plugin_*.rst` were generated with `plugintable.py`

A Universal RDF Store Interface

This document attempts to summarize some fundamental components of an RDF store. The motivation is to outline a standard set of interfaces for providing the support needed to persist an RDF Graph in a way that is universal and not tied to any specific implementation.

For the most part, the interface adheres to the core RDF model and uses terminology that is consistent with the RDF Model specifications. However, this suggested interface also extends an RDF store with additional requirements necessary to facilitate those aspects of Notation 3 that go beyond the RDF model to provide a framework for First Order Predicate Logic processing and persistence.

Terminology

Context

A named, unordered set of statements (that could also be called a sub-graph). The `named graph` literature and ontology are relevant to this concept. The term `context` could be thought of as either the sub-graph itself or the relationship between an RDF triple and a sub-graph in which it is found (this latter is how the term context is used in the Notation 3 Design Issues page).
It is worth noting that the concept of logically grouping triples within an addressable 'set' or 'subgraph' is just barely beyond the scope of the RDF model. The RDF model defines a graph to be an arbitrary collection of triples and the semantics of these triples — but doesn't give guidance on how to address such arbitrary collections in a consistent manner. Although a collection of triples can be thought of as a resource itself, the association between a triple and the collection (of which it is a part) is not covered. Public RDF is an example of an attempt to formally model this relationship - and includes one other unrelated extension: Articulated Text

Conjunctive Graph

This refers to the 'top-level' Graph. It is the aggregation of all the contexts within it and is also the appropriate, absolute boundary for closed world assumptions / models. This distinction is the low-hanging fruit of RDF along the path to the semantic web and most of its value is in (corporate/enterprise) real-world problems:

> There are at least two situations where the closed world assumption is used. The first is where it is assumed that a knowledge base contains all relevant facts. This is common in corporate databases. That is, the information it contains is assumed to be complete

From a store perspective, closed world assumptions also provide the benefit of better query response times, due to the explicit closed world boundaries. Closed world boundaries can be made transparent by federated queries that assume each `ConjunctiveGraph` is a section of a larger, unbounded universe. So a closed world assumption does not preclude you from an open world assumption.
For the sake of persistence, Conjunctive Graphs must be distinguished by identifiers (which may not necessarily be RDF identifiers or may be an RDF identifier normalized - SHA1/MD5 perhaps - for database naming purposes) that could be referenced to indicate conjunctive queries (queries made across the entire conjunctive graph) or appear as nodes in asserted statements. In this latter case, such statements could be interpreted as being made about the entire 'known' universe. For example:

```
<urn:uuid:conjunctive-graph-foo> rdf:type :ConjunctiveGraph
<urn:uuid:conjunctive-graph-foo> rdf:type log:Truth
<urn:uuid:conjunctive-graph-foo> :persistedBy :MySQL
```

Quoted Statement

A statement that isn't asserted but is referred to in some manner. Most often, this happens when we want to make a statement about another statement (or set of statements) without necessarily saying these quoted statements (are true). For example:

```
Chimezie said "higher-order statements are complicated"
```

Which can be written (in N3) as:

```
:chimezie :said {:higherOrderStatements rdf:type :complicated}
```

Formula

A context whose statements are quoted or hypothetical.
Context quoting can be thought of as very similar to reification. The main difference is that quoted statements are not asserted or considered as statements of truth about the universe and can be referenced as a group: a hypothetical RDF Graph

Universal Quantifiers / Variables

(relevant references):

- OWL Definition of SWRL.
- SWRL/RuleML Variable

Terms

Terms are the kinds of objects that can appear in a quoted/asserted triple.
This includes those that are core to RDF:

- Blank Nodes
- URI References
- Literals (which consist of a literal value, datatype and language tag)

Those that extend the RDF model into N3:

- Formulae
- Universal Quantifications (Variables)

And those that are primarily for matching against 'Nodes' in the underlying Graph:

- REGEX Expressions
- Date Ranges
- Numerical Ranges

Nodes

Nodes are a subset of the Terms that the underlying store actually persists. The set of such Terms depends on whether or not the store is formula-aware. Stores that aren't formula-aware would only persist those terms core to the RDF Model, and those that are formula-aware would be able to persist the N3 extensions as well. However, utility terms that only serve the purpose for matching nodes by term-patterns probably will only be terms and not nodes.
The set of nodes of an RDF graph is the set of subjects and objects of triples in the graph.

Context-aware

An RDF store capable of storing statements within contexts is considered context-aware. Essentially, such a store is able to partition the RDF model it represents into individual, named, and addressable sub-graphs.

Formula-aware

An RDF store capable of distinguishing between statements that are asserted and statements that are quoted is considered formula-aware.
Such a store is responsible for maintaining this separation and ensuring that queries against the entire model (the aggregation of all the contexts - specified by not limiting a 'query' to a specifically name context) do not include quoted statements. Also, it is responsible for distinguishing universal quantifiers (variables).

Note: These 2 additional concepts (formulae and variables) must be thought of as core extensions and distinguishable from the other terms of a triple (for the sake of the persistence round trip - at the very least). It's worth noting that the 'scope' of universal quantifiers (variables) and existential quantifiers (BNodes) is the formula (or context - to be specific) in which their statements reside. Beyond this, a Formula-aware store behaves the same as a Context-aware store.

Conjunctive Query

Any query that doesn't limit the store to search within a named context only. Such a query expects a context-aware store to search the entire asserted universe (the conjunctive graph). A formula-aware store is expected not to include quoted statements when matching such a query.

N3 Round Trip

This refers to the requirements on a formula-aware RDF store's persistence mechanism necessary for it to be properly populated by a N3 parser and rendered as syntax by a N3 serializer.

Transactional Store

An RDF store capable of providing transactional integrity to the RDF operations performed on it.

Interpreting Syntax

The following Notation 3 document:

```
{ ?x a :N3Programmer } => { ?x :has [a :Migraine] }
```

Could cause the following statements to be asserted in the store:

```
_:a log:implies _:b
```

This statement would be asserted in the partition associated with quoted statements (in a formula named `_:a`)

```
?x rdf:type :N3Programmer
```

Finally, these statements would be asserted in the same partition (in a formula named `_:b`)

```
?x :has _:c

_:c rdf:type :Migraine
```

Formulae and Variables as Terms

Formulae and variables are distinguishable from URI references, Literals, and BNodes by the following syntax:

```
{ .. } - Formula ?x - Variable
```

They must also be distinguishable in persistence to ensure they can be round-tripped.

Note: There are a number of other issues regarding the **:doc:'persisting of N3 terms <persisting_n3_terms>'_**.

Database Management

An RDF store should provide standard interfaces for the management of database connections. Such interfaces are standard to most database management systems (Oracle, MySQL, Berkeley DB, Postgres, etc..)

The following methods are defined to provide this capability (see below for description of the *configuration* string):

`Store.`**`open`**(*configuration*, *create=False*)
: Opens the store specified by the configuration string. If create is True a store will be created if it does not already exist. If create is False and a store does not already exist an exception is raised. An exception is also raised if a store exists, but there is insufficient permissions to open the store. This should return one of: VALID_STORE, CORRUPTED_STORE, or NO_STORE

`Store.`**`close`**(*commit_pending_transaction=False*)
: This closes the database connection. The commit_pending_transaction parameter specifies whether to commit all pending transactions before closing (if the store is transactional).

`Store.`**`destroy`**(*configuration*)
: This destroys the instance of the store identified by the configuration string.

The *configuration* string is understood by the store implementation and represents all the parameters needed to locate an individual instance of a store. This could be similar to an ODBC string or in fact be an ODBC string, if the connection protocol to the underlying database is ODBC.

The *open()* function needs to fail intelligently in order to clearly express that a store (identified by the given configuration string) already exists or that there is no store (at the location specified by the configuration string) depending on the value of `create`.

Triple Interfaces

An RDF store could provide a standard set of interfaces for the manipulation, management, and/or retrieval of its contained triples (asserted or quoted):

`Store.`**`add`**(*(subject, predicate, object)*, *context*, *quoted=False*)
: Adds the given statement to a specific context or to the model. The quoted argument is interpreted by formula-aware stores to indicate this statement is quoted/hypothetical It should be an error to not specify a context and have the quoted argument be True. It should also be an error for the quoted argument to be True when the store is not formula-aware.

`Store.`**`remove`**(*(subject, predicate, object)*, *context=None*)
: Remove the set of triples matching the pattern from the store

`Store.`**`triples`**(*triple_pattern*, *context=None*)
: A generator over all the triples matching the pattern. Pattern can include any objects for used for comparing against nodes in the store, for example, REGEXTerm, URIRef, Literal, BNode, Variable, Graph, QuotedGraph, Date? DateRange?

Parameters **context** – A conjunctive query can be indicated by either

providing a value of None, or a specific context can be queries by passing a Graph instance (if store is context aware).

Note: The *triples()* method can be thought of as the primary mechanism for producing triples with nodes that match the corresponding terms in the `(s, p, o)` term pattern provided. The term pattern `(None, None, None)` matches *all* nodes.

`Store.__len__`(*context=None*)

Number of statements in the store. This should only account for non- quoted (asserted) statements if the context is not specified, otherwise it should return the number of statements in the formula or context given.

Parameters **context** – a graph instance to query or None

Formula / Context Interfaces

These interfaces work on contexts and formulae (for stores that are formula-aware) interchangeably.

`ConjunctiveGraph.contexts`(*triple=None*)

Iterate over all contexts in the graph

If triple is specified, iterate over all contexts the triple is in.

`ConjunctiveGraph.remove_context`(*context*)

Removes the given context from the graph

Interface Test Cases

Basic

Tests parsing, triple patterns, triple pattern removes, size, contextual removes

Source Graph

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix : <http://test/> .
{:a :b :c; a :foo} => {:a :d :c} .
_:foo a rdfs:Class .
:a :d :c .
```

Test code

```
implies = URIRef("http://www.w3.org/2000/10/swap/log#implies")
a = URIRef('http://test/a')
b = URIRef('http://test/b')
c = URIRef('http://test/c')
d = URIRef('http://test/d')
for s,p,o in g.triples((None,implies,None)):
    formulaA = s
    formulaB = o
```

```
    #contexts test
    assert len(list(g.contexts()))==3

    #contexts (with triple) test
    assert len(list(g.contexts((a,d,c))))==2

    #triples test cases
    assert type(list(g.triples((None,RDF.type,RDFS.Class)))[0][0]) == BNode
    assert len(list(g.triples((None,implies,None))))==1
    assert len(list(g.triples((None,RDF.type,None))))==3
    assert len(list(g.triples((None,RDF.type,None),formulaA)))==1
    assert len(list(g.triples((None,None,None),formulaA)))==2
    assert len(list(g.triples((None,None,None),formulaB)))==1
    assert len(list(g.triples((None,None,None))))==5
    assert len(list(g.triples((None,URIRef('http://test/d'),None),formulaB)))==1
    assert len(list(g.triples((None,URIRef('http://test/d'),None))))==1

    #Remove test cases
    g.remove((None,implies,None))
    assert len(list(g.triples((None,implies,None))))==0
    assert len(list(g.triples((None,None,None),formulaA)))==2
    assert len(list(g.triples((None,None,None),formulaB)))==1
    g.remove((None,b,None),formulaA)
    assert len(list(g.triples((None,None,None),formulaA)))==1
    g.remove((None,RDF.type,None),formulaA)
    assert len(list(g.triples((None,None,None),formulaA)))==0
    g.remove((None,RDF.type,RDFS.Class))

    #remove_context tests
    formulaBContext=Context(g,formulaB)
    g.remove_context(formulaB)
    assert len(list(g.triples((None,RDF.type,None))))==2
    assert len(g)==3 assert len(formulaBContext)==0
    g.remove((None,None,None))
    assert len(g)==0
```

Formula and Variables Test

Source Graph

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs:  <http://www.w3.org/2000/01/rdf-schema#> .
@prefix : <http://test/> .
{?x a rdfs:Class} => {?x a :Klass} .
```

Test Code

```
implies = URIRef("http://www.w3.org/2000/10/swap/log#implies")
klass = URIRef('http://test/Klass')
for s,p,o in g.triples((None,implies,None)):
    formulaA = s
    formulaB = o
    assert type(formulaA) == Formula
    assert type(formulaB) == Formula
```

```
    for s,p,o in g.triples((None,RDF.type,RDFS.Class)),formulaA):
        assert type(s) == Variable
    for s,p,o in g.triples((None,RDF.type,klass)),formulaB):
        assert type(s) == Variable
```

Transactional Tests

To be instantiated.

Additional Terms to Model

These are a list of additional kinds of RDF terms (all of which are special Literals)

- `rdflib.plugins.store.regexmatching.REGEXTerm` - a REGEX string which can be used in any term slot in order to match by applying the Regular Expression to statements in the underlying graph.
- Date (could provide some utility functions for date manipulation / serialization, etc..)
- DateRange

Namespace Management Interfaces

The following namespace management interfaces (defined in Graph) could be implemented in the RDF store. Currently, they exist as stub methods of *`Store`* and are defined in the store subclasses (e.g. `IOMemory`):

`Store.`**`bind`**`(`*prefix*, *namespace*`)`

`Store.`**`prefix`**`(`*namespace*`)`

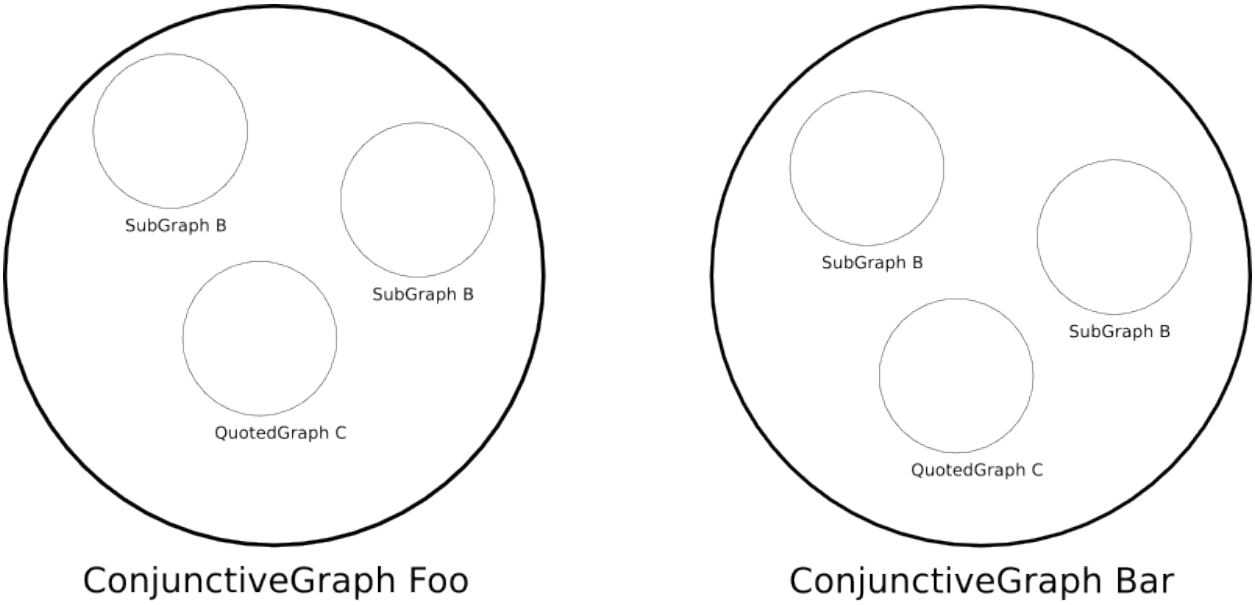
`Store.`**`namespace`**`(`*prefix*`)`

`Store.`**`namespaces`**`()`

Open issues

Does the Store interface need to have an identifier property or can we keep that at the Graph level?

The Store implementation needs a mechanism to distinguish between triples (quoted or asserted) in ConjunctiveGraphs (which are mutually exclusive universes in systems that make closed world assumptions - and queried separately). This is the separation that the store identifier provides. This is different from the name of a context within a ConjunctiveGraph (or the default context of a conjunctive graph). I tried to diagram the logical separation of ConjunctiveGraphs, SubGraphs and QuotedGraphs in this diagram



An identifier of `None` can be used to indicate the store (aka *all contexts*) in methods such as *`triples()`*, *`__len__()`*, etc. This works as long as we're only dealing with one Conjunctive Graph at a time – which may not always be the case.

Is there any value in persisting terms that lie outside N3 (`rdflib.plugins.store.regexmatching.REGEXTerm`, Date, etc..)?

Potentially, not sure yet.

Should a conjunctive query always return quads instead of triples? It would seem so, since knowing the context that produced a triple match is an essential aspect of query construction / optimization. Or if having the triples function yield/produce different length tuples is problematic, could an additional - and slightly redundant - interface be introduced?:

`ConjunctiveGraph`.**`quads`**(*triple_or_quad=None*)
Iterate over all the quads in the entire conjunctive graph

Stores that weren't context-aware could simply return `None` as the 4th item in the produced/yielded tuples or simply not support this interface.

Persisting Notation 3 Terms

Using N3 Syntax for Persistence

Blank Nodes, Literals, URI References, and Variables can be distinguished in persistence by relying on Notation 3 syntax convention.

All URI References can be expanded and persisted as:

```
<..URI..>
```

All Literals can be expanded and persisted as:

```
"..value.."@lang or "..value.."^^dtype_uri
```

Note: `@lang` is a language tag and `^^dtype_uri` is the URI of a data type associated with the Literal

Blank Nodes can be expanded and persisted as:

```
_:Id
```

Note: where Id is an identifier as determined by skolemization. Skolemization is a syntactic transformation routinely used in automatic inference systems in which existential variables are replaced by 'new' functions - function names not used elsewhere - applied to any enclosing universal variables. In RDF, Skolemization amounts to replacing every blank node in a graph by a 'new' name, i.e. a URI reference which is guaranteed to not occur anywhere else. In effect, it gives 'arbitrary' names to the anonymous entities whose existence was asserted by the use of blank nodes: the arbitrariness of the names ensures that nothing can be inferred that would not follow from the bare assertion of existence represented by the blank node. (Using a literal would not do. Literals are never 'new' in the required sense.)

Variables can be persisted as they appear in their serialization `(?varName)` - since they only need be unique within their scope (the context of their associated statements)

These syntactic conventions can facilitate term round-tripping.

Variables by Scope

Would an interface be needed in order to facilitate a quick way to aggregate all the variables in a scope (given by a formula identifier)? An interface such as:

```
def variables(formula_identifier)
```

The Need to Skolemize Formula Identifiers

It would seem reasonable to assume that a formula-aware store would assign Blank Node identifiers as names of formulae that appear in a N3 serialization. So for instance, the following bit of N3:

```
{?x a :N3Programmer} => {?x :has :Migrane}
```

Could be interpreted as the assertion of the following statement:

```
_:a log:implies _:b
```

However, how are `_:a` and `_:b` distinguished from other Blank Nodes? A formula-aware store would be expected to persist the first set of statements as quoted statements in a formula named `_:a` and the second set as quoted statements in a formula named `_:b`, but it would not be cost-effective for a serializer to have to query the store for all statements in a context named `_:a` in order to determine if `_:a` was associated with a formula (so that it could be serialized properly).

Relying on `log:Formula` Membership

The store could rely on explicit `log:Formula` membership (via `rdf:type` statements) to model the distinction of Blank Nodes associated with formulae. However, would these statements be expected from an N3 parser or known implicitly by the store? i.e., would all such Blank Nodes match the following pattern:

```
?formula rdf:type log:Formula
```

Relying on an Explicit Interface

A formula-aware store could also support the persistence of this distinction by implementing a method that returns an iterator over all the formulae in the store:

```
def formulae(triple=None)
```

This function would return all the Blank Node identifiers assigned to formulae or just those that contain statements matching the given triple pattern and would be the way a serializer determines if a term refers to a formula (in order to properly serializer it).

How much would such an interface reduce the need to model formulae terms as first class objects (perhaps to be returned by the `triple()` function)? Would it be more useful for the `Graph` (or the store itself) to return a Context object in place of a formula term (using the formulae interface to make this determination)?

Conversely, would these interfaces (variables and formulae) be considered optimizations only since you have the distinction by the kinds of terms triples returns (which would be expanded to include variables and formulae)?

Persisting Formula Identifiers

This is the most straight forward way to maintain this distinction - without relying on extra interfaces. Formula identifiers could be persisted distinctly from other terms by using the following notation:

```
{_:bnode} or {<.. URI ..>}
```

This would facilitate their persistence round-trip - same as the other terms that rely on N3 syntax to distinguish between each other.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

functional properties A functional property is a property that can have only one (unique) value y for each instance x, i.e. there cannot be two distinct values y1 and y2 such that the pairs (x,y1) and (x,y2) are both instances of this

> property. – http://www.w3.org/TR/owl-ref/#FunctionalProperty-def

graph An RDF graph is a set of RDF triples. The set of nodes of an RDF graph is the set of subjects and objects of triples in the graph.

named graph Named Graphs is the idea that having multiple RDF graphs in a single document/repository and naming them with URIs provides useful additional functionality. – http://www.w3.org/2004/03/trix/

transitivity A property is transitive:

> if whenever an element `a` is related to an element `b`, and `b` is in turn related to an element `c`, then `a` is also related to `c`. – http://en.wikipedia.org/wiki/Transitive_relation

Standard examples include `rdfs:subClassOf` or greater-than

e

r

Symbols

A

B

C

D

E

F

G

H

L

M

N

O

P

Q

R

S

T

U

V

W

X